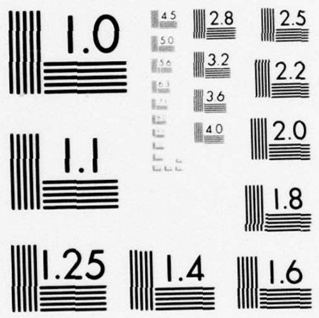
AD-A062 099　CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC F/G 9/2
CHEAP PRODUCTION OF JAPANESE DOCUMENTS, AN EXPERIMENT IN PROGRA--ETC(U)
JUN 78 I KIMURA　F44620-73-C-0074
UNCLASSIFIED　CMU-CS-78-130　AFOSR-TR-1495　NL

1 OF 1
AD
A062099

END
DATE
FILMED
3-79
DDC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Cheap Production of Japanese Documents, an Experiment in Programming Methodology*

Izumi Kimura**

Carnegie-Mellon University
and
Tokyo Institute of Technology

June 30, 1978

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

(14) CMU-CS-78-130

(6)

Cheap Production of Japanese Documents, an Experiment in Programming Methodology*

(10) Izumi Kimura**

Carnegie-Mellon University
and
Tokyo Institute of Technology

(15) F44620-73-C-0074, DAHC15-72-C-0038

(11) 30 June ~~30,~~ 1978

(12) 79p.

(18) AFOSR (19) TR-1495

----

* This work was supported in part by the Advanced Research Projects Agency of the Department of Defense under contracts DAHC 15-72-C-038 and F44620-73-C-0074 (which is monitored by the Air Force Office of Scientific Research), and in part by National Science Foundation Grant DCR 74-04187.

**Visiting CMU from T.I.T. Present address: Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa. 15213. From September 1978 on: Department of Information Science, Tokyo Institute of Technology, Ookayama, Meguroku, Tokyo 152, Japan.

403 081

Abstract

This paper describes a small experiment in programming methodology. The problem is to do something nice for the production of Japanese documents in a given environment. The assumed environment is that of the Department of Computer Science, Carnegie-Mellon University (CMU). The experiment is done by a one-man team consisting of the author. The process involves four factors: (1) preparing data, (2) finding the properties of the computing environment, (3) designing the user interface, and (4) actually writing a program. All these proceeds in parallel, and results in an inefficient but well-considered "mock-up", on which a more efficient production version can be based. The program, written in Snobol 4, accepts a sort of romanized Japanese. The output, printed on the Xerox Graphics Printer of CMU, makes mixed use of the hirakana and the katakana characters, but the kanji (Chinese characters) is excluded. At the focus of attention is how the general shape of the software is determined, i.e., requirement analysis in the broad sense. We try to support the developer's imagination. For this purpose we combine disciplined and undisciplined life-styles. Relations to the works of Sandewall, Kernighan and Plauger, and others are discussed. The first half of this paper also serves as a user's manual of the product.

Key Words and Phrases

Software engineering, programming methodology, requirement analysis, text processing, Japanese documents, manpower limitation, Snobol, Xerox Graphics Printer, controlled sloppiness, left-corner construction.

Chapter 1. INTRODUCTION

1.1. AN OUTLINE

This paper describes a small experiment in programming methodology. Given a rather fancy computing environment, and severely limited developer's time, a small system for producing Japanese documents is developed. We are interested in programming in an extremely broad sense. We expect at least four parts in it: (1) preparing data; (2) finding the properties of the computing environment; (3) designing the user interface; and (4) actually writing a program. Certainly, these four parts are essential in any data processing system. We wish to see how these factors interact, and how they can be controlled.

The computing environment assumed is that of the Computer Science Department, Carnegie-Mellon University (CMU) with its big PDP-10 and a special on-line printer called the Xerox Graphics Printer (XGP)[1]. The time spent is about one half of four months of the author (perhaps 400 man-hours), excepting the time for documentation per se. The resulting system handles only phonetic characters in a style often found in children's books. The more standard, adult's notation is found by far too costly for us. The program, written in Snobol 4, is a mock-up. It is big and slow, but such that you can actually use it to get experience. In writing the program, language-specific tricks have been avoided as far as possible. The program can be used, if desired, as a pattern for developing a more efficient version later.

In this experiment, we are particularly concerned with the conception stage of software development, i.e., requirement analysis (or requirement definition)[2] in the broad sense. Our focus of attention is to determine the general shape of software in such a way that the usefulness of the final product is maximized within the given environment.

The single most important criterion we choose for assessing our success or failure is whether the effort is a fun for us. The reader is urged not to misunderstand. The motive of this research is of course completely serious. The development of the general shape of software requires the developer's imagination. This is particularly true when the subject matter is not fully understood yet. The process must be a fun rather than a frustration, since otherwise the developer might subconsciously skip tiresome details.

The author's method somewhat resembles the Lisp user's life-style described by Sandewall[3]. In fact, experience indicates that their (in a sense) undisciplined method does have considerable merits of its own. We believe that our approach well compensates the drawbacks, which their method would have in a production programming environment. Our method is also related to "left-corner construction" of Kernighan and Plauger[4].

In the rest of this chapter, we shall say some more about our motivation (Section 1.2), the particular problem chosen for this case study (Sections 1.3 through 1.5), the computing environment (Section 1.6), and the scope and the nature of our solution (Section 1.7). Chapter 2 gives a manual-like description of our product. Chapter 3 examines how we reached it. Chapter 4 makes some additional remarks. The Appendix gives a short summary of this paper in Japanese.

This paper has been typeset by our system itself. Some manual postprocessing (numbering the pages and arranging the figures) has been necessary since our system is a mock-up, and has a number of unimplemented features (Section 2.3). There is a plan to fill these holes.

1.2. MOTIVATION

When "structured programming" and other modern programming ideas became widely known, words of abhorrence came from those who were developing software for their own research purposes in such areas as physics, chemistry, and automatic programming. They claimed that in research, programming could not be preplanned, and that imposing discipline would suppress freedom of thought. It was apparent that some of them were just trying to justify sloppiness for sloppiness' sake, but looking back, considerable wisdom is found in what they said at that time.

They do programming as parts of their research, a process of discovery. The results today could affect what they do tomorrow. If they are to wait until the problem becomes so well-defined that, say, a predicate can be written for specifying the product, they cannot start until their research is over. They program for something unknown. They must start anyway.

We have a similar situation in the conception stage of software development. We don't know exactly what the product would look like eventually. We cannot wait until everything becomes clear. We, too, must start anyway.

The question is, then: "How should we program for an unknown problem?" We will find an answer to this question through a case study.

1.3. THE READABILITY AND NON-WRITABILITY OF JAPANESE

We now talk about the problem area considered.

Documents in Japanese are known for high readability. This is certainly true with typeset documents. People can read very fast without special training for doing so. There are no fast-reading courses in Japan. Well-educated adults can read fast anyway.

For example, the first two sentences of the preceding paragraph would read

日本語の文書は読みやすいことで知られている。 活字で組んである場合、たしかにそういうことが言える。

Keywords such as 日本語 (the Japanese language) form compact units, and assist the reader to grasp the meaning very quickly. No spaces are used. They are not necessary. The boundaries of the words are apparent from the changes of the fonts.

The Japanese written language has a large character set. There are three classes of characters: (1) the HIRAKANA: about 50 syllabic characters for common use; (2) the KATAKANA: about 50 corresponding syllabic characters used mainly for representing imported Western words; and (3) the KANJI: more than 2,000 ideographic characters originally imported form ancient China. In technical articles, (4) roman alphabetic characters and (5) mathematical symbols are also used. (By KANA we mean both the hirakana and the katakana.)

In the usual practice, important concepts are represented by strings consisting solely of, or at least starting with, either the katakana(2) or the kanji(3), with the gaps filled by the hirakana(1) indicating grammatical relationships. (For example, 日本語 of the above consists solely of the kanji characters.) Because of their origins, the hirakana includes more curved strokes, while the katakana and the kanji essentially consists of straight line segments. They can be distinguished at a glance. Japanese documents are originally "underlined", so to speak, for important concepts.

This story, however, has a dark side: writing in Japanese is a very tedious and time-consuming job. The standard method is to write by hand on special sheets with preprinted boxes, filling one box with one character, and have the document typeset manually by a craftsman. Human labor involved is tremendous. Japanese scientists often grieve that they are definitely inferior to their Western colleagues in the quantity of pulp they are consuming. Their handicap is now becoming more prominent due to the advent of computerized document preparation. As far as technical articles are concerned, the author has less mental barrier when he writes in English than in Japanese even though the former language is not native for him. In English, he can do touch-typing, and edit the text within the computer.

In addition to scientific papers, letters cause difficulties. In writing a letter in English, we can ask a secretary for help. No such help is available for a letter in Japanese. Unless you are a V.I.P., you are supposed to write yourself by hand. (If you are one, trained secretaries will write very neatly, or type on a "Japanese" typewriter, Section 1.5.3, for you.) The process is again tedious, and time-consuming. Worse, there is a tradition in Japan that if you don't write neatly, people tend to doubt your sincerity. The Japanese people, even businessmen, often avoid letters, and prefer telephone calls.

In our case study, we shall try to do some nice things for helping the preparation of Japanese documents. In particular, we wish to help researchers in their everyday tasks.

The following two sections will give more information about the Japanese written language. This material is for the non-Japanese reader. The Japanese reader may wish to skip it.

1.4. THE HISTORICAL BACKGROUND

Originally, the Japanese language had no written representation. Characters were imported from ancient China, and, besides for writing Chinese documents, used for representing the Japanese language in phonetic approximations. The use culminated in the sixth century in a monumental anthology "Man'yo Shu", which recorded creations of all classes of people, from emperors to lowest-level soldiers, in that form.

Later, from the nineth centry on, shorthand phonetic notations for the Japanese syllables developed. There were two kinds, the hirakana and the katakana mentioned earlier.

The hirakana was derived from very quickly written forms of Chinese characters, and used mainly by women. One memorable event was that "The Tale of Genji" was written by Lady Murasaki from the late tenth to the early eleventh century. This great roman, known to the Western world by a translation of Arthur Waley, was written entirely in the hirakana.

The katakana, on the other hand, extracted some of the strokes from the Chinese characters. Its main usage was for putting memos on Chinese documents for facilitating the reading. Among the typical users were Buddhist priests, who had to chant sutra (Buddhist scriptures) in services.

Since then, the hirakana and the katakana coexisted with the kanji (Chinese characters). These three were mixed in various ways. The hirakana was held to be a more emotional and less prestigious script. The katakana, on the other hand, was used in more political and religious situations, usually in combination with the kanji, and in that form tended to be a prestigious script.

This situation has changed in the long run. The hirakana has become more standard. As noted earlier, the present practice is to use the hirakana as a glue for grammatically connecting important concepts represented mainly in the kanji and the katakana.

1.5. PAST ATTEMPTS OF MECHANIZATION

We shall now examine the past attempts of mechanizing the production of Japanese documents[5]. There are five classes:

(1) Romanized Japanese on ordinary typewriters;
(2) Katakana typewriters;
(3) "Japanese" typewriters, and kanji teleprinters;
(4) Hirakana typewriters;
(5) Recent computer-based systems.

1.5.1. ROMANIZED JAPANESE

The first solution is an obvious one. Early attempts of romanizing Japanese date back to the sixteenth century, when Catholic missionaries tried to record the Japanese spoken language of that time in the roman alphabet. Later, in the nineteenth century, an American missionary James Hepburn developed a system based on the English pronunciation. His system, after some modifications, became a standard. It is now called the Hepburn system of romanization.

Still later, other systems developed, namely, the "Japanese" and the official systems. Unlike the Hepburn system, which basically transcribed pronunciation, the later systems attempted to be more faithful to the grammatical structure of the Japanese language. (This considerably sacrificed the fitness of the notation to the English context.)

Romanized Japanese has the great advantage that ordinary typewriters can be used almost unmodified. Unfortunately, this notation is very redundant, and hard to read. One reason is that the words of Chinese origin, usually written in the kanji, often degenerate. For example, there are at least nine distinct words which are read "seisan". They are represented by different combinations of Chinese characters, and have different meanings: a formal dinner, production, prospect, cyanic acid, ghastliness, liquidation (of a company and the like), the age of an emperor, the Holy Communion, and an exact calculation. The recent tendency is to use these ambiguous words less and less. This big problem, however, is expected to remain for a long time to come.

For these reasons, romanized Japanese has not been accepted socially as an everyday tool. The editor of a scientific journal would never accept your paper if it is written in this style.

There is another unfortunate fact. The birth of the later systems caused a political controversy. There are differences of practice even within the Japanese government. The Ministry of Foreign Affairs insists on the use of the Hepburn system. Your name will be automatically spelled according to it in a passport. On the other hand, the Ministry of Education sticks to the official system. The employees of some of the research institutes belonging to the latter Ministry are supposed to publish papers with their names spelled according to the official system. (This has caused difficulties to the authors when they travel overseas. People in foreign countries sometimes doubt their identities.) Under these circumstances, it is safer to avoid writing letters in romanized Japanese. People might guess that you are a fanatic adherent of that particular system you use.

1.5.2. KATAKANA TYPEWRITERS

This second solution has a history of more than 50 years. The katakana was pushed into the ordinary typewriter. This solution requires spaces because without them the word boundaries would be very difficult to detect. This is in conflict with the standard practice. (Of course the romanized notation also requires spaces.) Moreover, just as with romanized Japanese, imported Chinese words cause problems.

Despite these difficulties, it was adopted by some forward-looking businessmen. This notation is at least better than the first one because it is less redundant, and the characters are native. Its typical use has long been in writing business slips. After the advent of computers, people began to print mail addresses also in that style. However, it is still rather redundant, and hard to read. It is acceptable only in in-house documents. Business letters are seldom written in the katakana. It cannot be an everyday tool of a scientist.

1.5.3. "JAPANESE" TYPEWRITERS AND KANJI TELEPRINTERS

The third solution attacks the problem from the opposite side. A "Japanese" typewriter has a matrix of types, including all classes of characters used in the Japanese language. The characters are selected from the matrix by operating a handle, and another handle is used to hit the type to cause the character to be printed. Even simplified models have thousands of entries in the matrix. Touch typing is impossible. Heavy training is necessary. Typing speed is of the order of one character per second. The result is very neat, and comfortable to read. Legal documents are often typed in this way. Recently, low-cost models became popular.

Some researchers are using them for preparing camera-ready versions of their papers, but one short paper often costs them one full day. Editing is impossible except by cumbersome means such as correction fluid and pasting over. Again, this cannot be an everyday tool.

A related subject is the kanji teleprinters. Early users include newspaper companies. In a typical system, two frames on a paper tape correspond to a character, which is chosen from a rotating drum, and printed at about the same rate as the "Japanese" typewriters are typed. Here, editing is possible, but the machine is bulky, and very expensive.

1.5.4. HIRAKANA TYPEWRITERS

The fourth solution is a recent addition. It is based on the observation that, after the invention of the katakana typewriter, the hirakana became much more common than the katakana. Basically, the hirakana typewriter is same as the katakana typewriter, but the types are replaced. The result is much easier to read.

The necessity of spaces and the degeneration of Chinese words are drawbacks which this solution shares with the first and the second solutions. The most uncomfortable aspect of this style lies in that the imported Western words must be spelled in the hirakana. They should rather be spelled in the katakana. The Western words spelled in this way give the impression that the writer is fooling the reader. (They are at least as unusual as "beethoven", "texas", or even "ibm".) Since scientific papers use Western words very often, there is little hope for this style to be tolerated by the editors of scientific journals. All in all, however, this is the least offending style among those based on ordinary typewriters. In fact, children's books are now usually written in this style.

1.5.5. RECENT COMPUTER-BESED SYSTEMS

Finally, the fifth solution. The main commercial usage is found in printing mail addresses. (Katakana addresses often cause errors in the post office, and are not polite to the recipients.) Various high-speed printing devices are available. Xerography, dot-printing, holography, and other technologies are used. The printers are still very expensive, but, according to the common understanding, the bottleneck is not in the output but in the input. A classical solution is to use the kanji teleprinter input device (the third solution), but this requires roughly one second per character. A special proprietary touch-typing method using an ordinary typewriter keyboard has been developed, and is said to attain a 200 character per minute rate, but the method requires a hard training. As many nonsense combinations of key strokes as are the Japanese characters must be memorized, though efforts have been made to assist the memory by association. Research is also going on for automatic kana-to-kanji conversion. Although some success has been reported, perfect conversion appears to be almost impossible. The biggest reason is the degeneration of the words of Chinese origin (discussed earlier). Remember that there are nine "seisan"s. Once more, this solution is, for now, far from providing a cheap everyday tool which an average researcher can afford.

In passing we note that a Japanese Industrial Standard (JIS) has recently been issued on the encoding of Japanese general characters including the kanji. In the author's opinion, this is one great step ahead.

1.6. THE COMPUTING ENVIRONMENT

This research began when the author found himself in a fancy computing environment of Carnegie-Mellon University(CMU), Department of Computer Science. In particular, there was a special printer called the Xerox Graphics Printer (XGP). This device can print various fonts and figures. The author felt a strong incentive toward doing something about Japanese document production using this device. We now briefly describe the pertinent part of the environment.

The XGP of CMU[1] (the printer itself) is, in a grossly oversimplified description, something like an ordinary Xerox copier in which images created by a computer-controlled cathode-ray tube is used in place of the reflection from an original document. It has a printing drum rotating at a constant speed. Dots are deposited on it typically with the density of 183 dots per inch, both lengthwise and breadthwise. The printer uses a 8.5 inch-wide roll of ordinary paper, which advances at the rate of about 1 inch per second. A computer-controlled paper cutter can be used to cut the paper at, say, every 11 inches.

The printer is controlled by a dedicated DEC PDP-11/45. In our mode of usage, the bit table area for storing font information has the capacity of approximately 15K words (1 word = 16 bits). (The core has 28K words in all.) Usually, the font information is first stored in a 256K fixed-head disk, and is loaded to the core as required. The number of dots which must be handled is so large that even with this powerful dedicated machine the limitation of its power often causes frustration.

The PDP-11 computer is in turn coupled to a general-purpose PDP-10 (a KI-10). The user operates the XGP by feeding commands to a program called LOOK[1a] runing on the PDP-10. In the application of this paper, the commands might look like

```
.r look
*do ship hira30/538
```

```
*do ship kata30/539
*text.xgo
```

The first line activates LOOK. The second and the third lines instruct that font information for the hirakana and the katakana be sent from the files "hira30" and "kata30" of the PDP-10 to the internal disk of the PDP-11 as fonts Nos.538 and 539, respectively. The fourth line instructs to send the text file "text.xgo" from the PDP-10 to the PDP-11.

There are much more details. For example, the PDP-10 connected to the PDP-11, called the B-system, is not for the general use. It is primarily for artificial intelligence research (and in particular, Speech and Vision research). Accordingly, the non-AI users must first log in to another PDP-10 (the A-system, a KL-10), and gain access to the B-system by a special trick called cross-patching. We shall avoid to enter into installation-specific details by sometimes telling a lie.

Due to the limitation of the PDP-11's main storage, LOOK can simultaneously handle only two fonts, each of which can contain no more than 127 characters. The characters belonging to the two fonts may be freely mixed. If more variation is desired, the fonts must be swapped with others in the disk. The process requires *some time*, during which the printing drum of the XGP advances. For this reason, a line can contain only those characters belonging to the two currently loaded pair of fonts. The two in-core fonts are called the A- and B-fonts.

LOOK interprets control codes embedded in the text file, and performs various functions such as choosing one of the two fonts, swapping the fonts, underlining, and overprinting.

More particularly, the rubout code (octal 177) serves as an escape code. The next character (code character) determines the action. For example, a Carriage Return code (octal 15) causes the B-font to be used. Currently, octal 0 through 71 are used for code characters. A number of characters that follow sometimes act as a parameter to the control code.

In the normal usage, the user does not generate a LOOK-readable text file directly. Instead, he uses a text editor to generate a mixture of the printed text and certain commands. It is then converted by a suitable service program into a LOOK-readable form. When this project was begun, a program called PUB[6] was standard for the conversion, but it was rather hard to use, poorly documented, and very slow (partly because it was based on macro processing). Later, a much faster and cleaner conversion program called SCRIBE was announced by Brian Reid[7], but was not available when we started.

Our project involved the task of developing Japanese fonts. Fortunately, this could be done rather pleasantly with the help of a graphic display and a font editor called BILOS[8].

1.7. A PREVIEW OF THE SOLUTION - MIXED KANA-ENGLISH DOCUMENTS

We very much wished to build a system which produces the standard mixed kanji-kana documents, but its impossibility was almost apparent from the outset. What we built was, so to speak, a combination of all the previous solutions based on ordinary typewriters (Solutions 1, 2, and 4 of Section 1.5.) In our notation, the sample sentences of Section 1.3 talking about the readability of the Japanese written language read

> にほんご の ドキュメント は よみやすい こと で しられて いる。 かつじ で くんで ある ばあい、たしか に そういう こと が いえる。

Here, we made a change in the choice of words in order to demonstrate the mixed use of the hirakana and the katakana. The English word "document", previously translated into a word of Chinese origin, has now been translated into a phonetic transcription (in the katakana) of the Western word, though some people hate these "katakanized" terms (just as Frenchmen are famous for hating imported American words). Words of Chinese origin are preferred. (Ironically, they are also imported words.)

We trust that even the non-Japanese reader will recognize the word in question. It is ドキュメント. Its non-Western counterpart

文書

can also be spelled in the hirakana as ぶんしょ, although this latter notation does not stand out, and appears somewhat childish for a Japanese eye. Thus, still another (childish) style we can produce by our system is

> にほんご の ぶんしょ は よみやすい こと で しられて いる。 かつじ で くんで ある ばあい、たしか に そういう こと が いえる。

More detailed discussion of our product will be made in the next chapter.

Chapter 2. A DESCRIPTION OF THE PRODUCT

In this chapter, we describe our product. Section 2.1 gives an overview. Section 2.2 gives a more complete user's view of the system. Possible enhancements are discussed in Section 2.3. We emphasize that what we describe here is a half product, i.e., a mock-up. The central issue of this paper is how to determine the general shape of the software. The present form is sufficient for this purpose, but for arriving at a production quality software we must do more.

2.1. A QUICK INTRODUCTION TO THE USER'S VIEW

Let us first give a brief, help-message-like description of our system. We note that, in the operating system we use, the uppercase and the lowercase characters have the same meanings as far as the monitor commands are concerned.

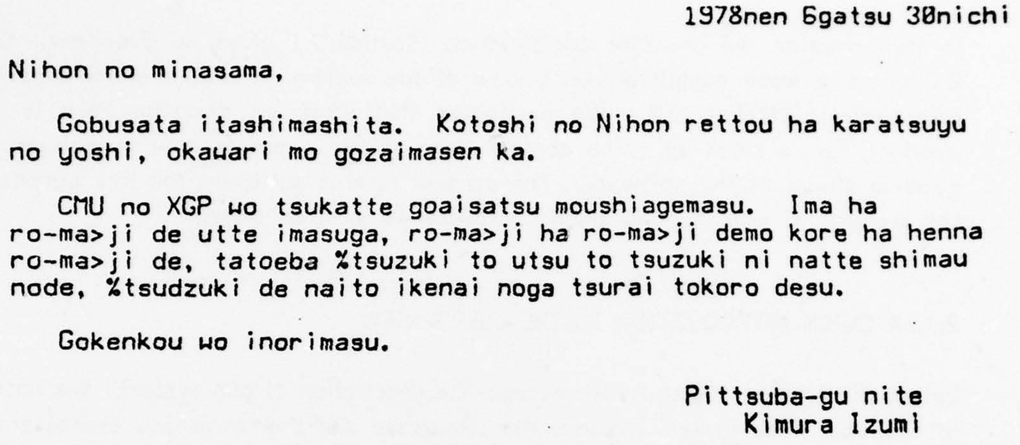
The user prepares his text in a sort of romanized Japanese. See FIG.1 for a sample input. This is a hypothetical letter from the author to his Japanese friends. The format basically follows the Hepburn system, but whenever possible, other systems are also accepted. The user is requested to keep the kana representation in mind. Thus, he should type "ha" for the case-defining particle spelled "wa" in the standard systems of romanization, and "wo" for that one spelled "o". Also type "ookii" (big), and "oyakoukou" (being kind to one's parents). Other somewhat unusual examples are:

tsudzuki	for つづき (continuation)
Be-to-ven	for ベートーヴェン (Beethoven)
sofutowwea	for ソフトウェア (software)

Thus, elongation of the vowels is represented by a minus sign. The user must type "wwe" in "software" because we wish to reserve "we" for the special character ゑ used in the classics. In fact, some Japanese linguists actually pronounce ゑ as "we" in order to distinguish it from usual え ("e"). The conversion from the romanized to the kana representation (heretofore termed ROMAN-KANA CONVERSION) is driven by a table of a self-explanatory format. The user may wish to change it.

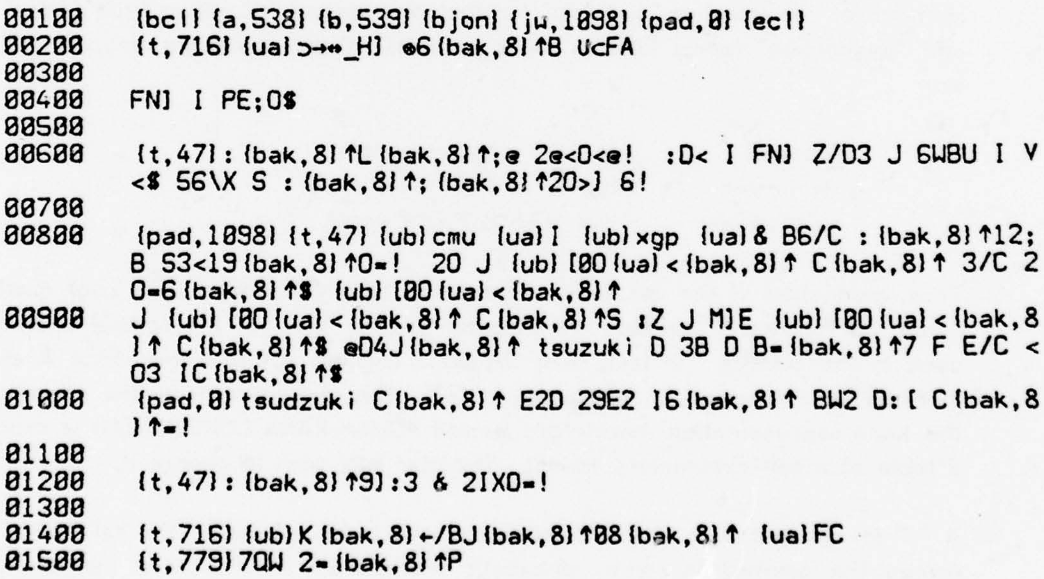
A "<" forces that part of the word following it to be printed in the katakana. A "7" forces the printing in roman alphabetic characters. A ">" forces the use of the hirakana. These "shift codes" are effective only up to the next shift code, a space, or the end of a line. However, a double occurrence of these shift codes, like "<<", acts as a global shift code, and forces everything up to the next shift code to be printed (whenever possible) in the designated case.

The user may omit some of the shift codes. A syllable is printed in the hirakana



```
                                                 1978nen 6gatsu 30nichi
Nihon no minasama,

   Gobusata itashimashita.  Kotoshi no Nihon rettou ha karatsuuyu
no yoshi, okawari mo gozaimasen ka.

   CMU no XGP wo tsukatte goaisatsu moushiagemasu.  Ima ha
ro-ma>ji de utte imasuga, ro-ma>ji ha ro-ma>ji demo kore ha henna
ro-ma>ji de, tatoeba %tsuzuki to utsu to tsuzuki ni natte shimau
node, %tsudzuki de naito ikenai noga tsurai tokoro desu.

   Gokenkou wo inorimasu.

                                             Pittsuba-gu nite
                                               Kimura Izumi
```

Fig.1: A sample input.



```
00100    {bcl} {a,538} {b,539} {bjon} {jw,1098} {pad,0} {ecl}
00200    {t,716} {ua}⊃→↔_H] ⊗6{bak,8}↑B U⊂FA
00300
00400    FN] I PE;O$
00500
00600    {t,47}: {bak,8}↑L{bak,8}↑;@ 2@<O<@!  :D< I FN] Z/D3 J 6WBU I V
         <$ 56\X S : {bak,8}↑; {bak,8}↑2O>] 6!
00700
00800    {pad,1098} {t,47} {ub}cmu {ua}I {ub}xgp {ua}& B6/C : {bak,8}↑12;
         B S3<19{bak,8}↑O=!  2O J {ub} [00{ua}<{bak,8}↑ C{bak,8}↑ 3/C 2
         O=6{bak,8}↑$ {ub} [00{ua}<{bak,8}↑
00900    J {ub} [00{ua}<{bak,8}↑ C{bak,8}↑S :Z J M]E {ub} [00{ua}<{bak,8
         }↑ C{bak,8}↑$ @D4J{bak,8}↑ tsuzuki D 3B D B={bak,8}↑7 F E/C <
         O3 IC{bak,8}↑$
01000    {pad,0} tsudzuki C{bak,8}↑ E2D 29E2 ]6{bak,8}↑ BW2 D:[ C{bak,8
         }↑=!
01100
01200    {t,47}: {bak,8}↑9]:3 & 2IXO=!
01300
01400    {t,716} {ub}K{bak,8}←/BJ{bak,8}↑08{bak,8}↑ {ua}FC
01500    {t,779}7QW 2={bak,8}↑P
```

Fig.2: An intermediate result (a JPN-file) corresponding to Fig.1.

unless there is a reason otherwise. However, "Pittsuba-gu" for "Pittsburgh", printed in the katakana, requires no shift code. The rule is that those syllables containing, in the converted form, one of the small "a", "i", "u", "e", "o" ("アイウエオ"), or, in the original romanized form, either an elongation sign (-) or a "p" not preceded by another "p" or "n" (or "m"), are printed in the katakana unless otherwise forced by a shift code. Thus, "Pittsuba-gu" contains an elongation (-), and begins with a "p". Those words, which are impossible to interpret as romanized Japanese words (such as "CMU"), are printed in the roman alphabets. The ASCII symbols are always printed as such. The exceptions are single quotes, double quotes, periods, commas, and minus signs. If they are in the Japanese context, they are suitably replaced by Japanese punctuation symbols or otherwise used during the conversion.

The user should type his text in such a way that its general shape indicates the format in which he wishes the final output to appear. The system makes guesses based on the assumption that you are typing for a 69-character line. On the basis of these guesses it attempts to translate the text paragraphs into right-justified text paragraphs, and the title lines into suitably indented title lines. Again, the 69-character assumption can be changed easily.

Presently, the conversion is done by a Snobol program. Although the program can be renamed in any arbitrary way, let us assume that the program is in the file "MATOME.SNO". (Matome means "summing up" in Japanese.) The user types

```
.r sitbol
*matome
```

The dot and the star are prompts from the system (DEC System-10 Monitor and the SITBOL system, respectively). Since our program is big, and the Snobol dialect SITBOL assumed here is not very fast, there will be some time before you get a prompt

```
Input filename or [CR] or ? :
```

You give the name of the file containing the romanized text (followed by a Carriage Return). Assume that the name of the file is "TEXT". Then our program creates two files for you: TEMP:TEXT.JPN, and TEMP:TEXT.XGO. The former (called a JPN-file) contains an intermediate result, which you can inspect if you wish. See Fig.2 for a sample JPN-file corresponding to Fig.1. The JPN-files have long lines. "00100", "00200", ... are line numbers added manually to show the beginnings of the lines. This figure contains some non-ASCII symbols. They correspond to additional characters with code values less than octal 40 (Section 2.2.10).

The latter (called an XGO-file) can be sent to the XGP. Since our program is

Snobol-based, and since no serious effort for speeding-up has been made, the process usually takes a long time. It prints a "*" for every ten lines of the input processed. Assuming that the file TEXT has 50 lines, the printout on your terminal might be

```
        from TEXT to TEMP:TEXT.JPN...*****
          from TEMP:TEXT.JPN to TEMP:TEXT.XGO...****
```

(The second half of the processing usually gives fewer *'s because the filling of the text paragraphs usually results in less output lines.) The system then returns with another prompt

```
Input filename or [CR] or ? :
```

You may repeat the process as often as you wish. You may wish to type "?" (followed by a CR) to get a help message. When you are through, just hit CR, which terminates the execution of the Snobol program. The computer returns to the SITBOL system, and prompts you by a star. You will usually respond by a control-C to return to the monitor, but you may alternatively activate another Snobol program.

You can now print the converted result by sending it to the XGP. Let us assume for simplicity that you are on the B-system (see Section 1.6). You invoke LOOK by telling the monitor as follows:

```
.r look
```

The system will answer you as

```
LOOK: Version .....
Type HELP for help
*
```

First you must load the Japanese fonts to the PDP-11's disk (if it is not already loaded). You answer to the prompt(*) as

```
*@cmd[c300ik40]
```

Here, we assume that a special user having the account number C300IK40 on the B-system has a special file called CMD, which contains commands for causing necessary font information to be "shipped" to the PDP-11. (CMD contains something like the second and the third lines of the sample command sequence of Section 1.6.) After the intended action is finished, you are told

```
**<end of command file>
*
```

Sometimes the system hangs up during the transmission of the font information for some reasons unknown to the author. In that case escape to the monitor state, and retry from the activation of LOOK on. In the second time, already-shipped font files are skipped without transmission. Assuming that you successfully reached the end of the command file, you type the name(s) of the file(s) you wish to print:

```
*text.xgo
```

(If you have more than one file, delimit the names by commas, or repeat the process for another "*" from the computer.) You will get a printout on the XGP. A sample output corresponding to Figs.1 and 2 is given in Fig.3. When you are through, answer to "*" with a control-C, to return to the monitor.

Implementation-specific remarks.

1. Since our system involves the use of TEMP, the user is recommended to declare this fact to the monitor in advance as follows:

   ```
   .mount temp
   ```

2. TEMP files are automatically purged if they remain unused for about two weeks. In the assumed environment, it is customary to put derived data such as a compiled program in the temp area (structure).

2.1.1. NOTE - THE A-SYSTEM USER'S VIEW

This subsection is only for a CMU user. Other readers should skip it.

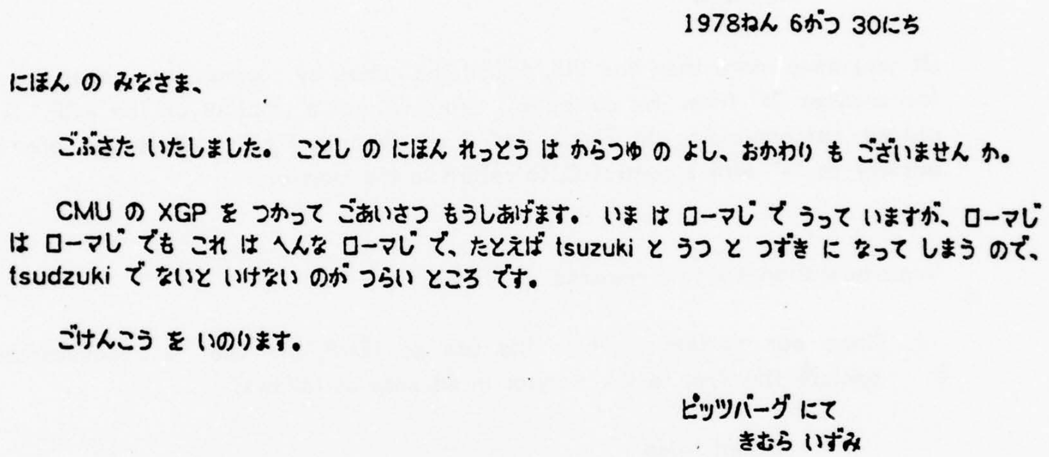
If you are on the A-system, you must invoke a program called BOOK, a special LOOK invoker, as follows:

```
.r book
```

You are prompted as

```
Type /H for help
Files:
```

You answer by typing the names of the files you wish to print, with the "extension" of the form ".xgo" implied. In this case you should just type "text", and hit CR.



1978ねん 6がつ 30にち

にほん の みなさま、

ごぶさた いたしました。 ことし の にほん れっとう は からつゆ の よし、おかわり も ございません か。

CMU の XGP を つかって ごあいさつ もうしあげます。 いま は ローマじ で うって いますが、ローマじ は ローマじ でも これ は へんな ローマじ で、たとえば tsuzuki と うつ と つずき に なって しまう ので、tsudzuki で ないと いけない のが つらい ところ です。

ごけんこう を いのります。

ピッツバーグ にて
きむら いずみ

Fig.3: An XGP printout corresponding to Fig.1.

This causes a trick called cross-patching to take you temporarily to the B-system, in which you will be prompted eventually as

```
*<You are now cross-patched.>
*
```

You can now behave as though you are using LOOK on the B-system, with the extension ".xgo" assumed in some cases. Type

```
*@cmd[c300ik40]
```

After proper response is received from the system, give the name(s) of your file(s), with ".xgo" implied, as

```
*text
```

When everything is finished, type control-C to return to the controller of BOOK, and tell it "q" for "quit" to return to the A-system. (If a hung-up occurs during the transmission of the fonts, type control-C, and tell the controller "r" for "restart".)

2.2. A MORE COMPLETE GUIDE

We believe that the above brief explanation is enough for enabling the user to begin experimenting with our system, but it is certainly incomplete. He may raise numerous questions. For example, he might ask about the meaning of the following (mysterious) sentence: "The user should type his text in such a way that its general shape indicates the format in which he wishes the final output to appear." In this section, we give more information so that the user can answer these questions himself.

2.2.1. THE CONCEPTUAL SYSTEM STRUCTURE

Our conversion program conceptually forms a chain of data-transforming modules. These are

a. Discriminator which looks at the general shape of the text, and locates text paragraphs and independent title lines. The input is taken from the user-supplied input file, and is a sequence of text lines possibly mixed with page marks (dividing the input file into logical pages). The output of this module is a sequence of TEXT BLOCKs, each of which may be an independent title line, a paragraph (consisting of glued text lines), or a page mark (dividing the output into pages).

b. Roman-Kana Converter which extracts syllables from the text blocks. Unless instructed otherwise by a shift code, and if the syllable is sensible as a romanized Japanese, it is converted into a kana representation. The output is CONVERTED TEXT BLOCKs.

c. Paragraph Cutter which calculates the widths of the characters, and arrange the paragraph lines to include just as many number of words that can be accommodated within a line. This module also arranges independent title lines. Right-justification is done by LOOK. Necessary control codes are supplied by this module.

d. Assembler, which converts the symbolic control codes into a LOOK-readable form.

Section 2.2.2 describes the Discriminator. The mystery mentioned above is cleared. Section 2.2.3 through 2.2.5 describe the Roman-Kana Converter. Sections 2.2.6 and 2.2.7 describe the Paragraph Cutter and the Assembler, respectively. Sections 2.2.8 through 2.2.10 provide some more information.

Remarks.

1. The above reflects a simplified mock-up as of this writing. As noted earlier, some additions are expected. For example, between (c) and (d), we expect a paginator module.

2. In the actual Snobol program, the modules (a)-(c) have been implemented by nested calls of functions. They are combined indivisibly. However, the output from the Paragraph Cutter is given explicitly in a file. The user can monitor it, or even modify it if he wishes. An example of this intermediate result has been given in Fig.2.

2.2.2. THE DISCRIMINATOR

The design of this module is an extension of a series of designs given in [10] and [11]. It attempts to save the user from the tedium of typing, as in [6],

```
.fill; adjust; indent 10,10,5;
```

or, as in [7],

```
@enter(example,group)
```

It infers the user's intention from the general shape of the input text, and formats the output accordingly. It cannot do very fancy things, but can do most of those services a researcher would need in his daily business. (His publisher will need fancier formatters, but it is another matter.)

The module reads input lines from a file, locates text paragraphs by an algorithm described below, glues the lines of each text paragraph to form a block, and sends these blocks to the output. The input may also contain independent title lines and page marks. They are sent to the output with no modification. Each paragraph block sent to the output contains a text string and an integer LEFTMAR. In the following description of the algorithm, we assume that the end-of-file condition and the page marks are received from the input file as though they are special kinds of input lines.

In our algorithm, we look at the pairs of adjacent lines (call them CURRENT and NEXT). The trailing spaces of the (ordinary text) lines are TRIMmed before being processed.

If CURRENT is an EOF, the processing ends. If CURRENT is a page mark, it is sent to the output, the old NEXT replaces CURRENT, and a new NEXT is obtained from the input file.

Otherwise let the numbers of leading spaces of CURRENT and NEXT be CMAR and NMAR, respectively. (If the line is empty, assume that it has 99999 leading spaces.) We check whether

1. NEXT exists in the same page. (That is, CURRENT is not the last line in the file, and there is no form feed between CURRENT and NEXT.)
2. Either CURRENT or NEXT has at least 40 (blank or nonblank) characters.
3. If $CMAR \geq NMAR$, then
   $NMAR \leq REFCOL + 4$, and
   $CMAR \leq NMAR + 11$.
4. If $CMAR < NMAR$, then
   $CMAR \leq REFCOL + 4$, and
   $NMAR \leq CMAR + 7$.

Condition 3 corresponds to a normal indented paragraph (including the blocked form paragraph as a special case), and Condition 4 corresponds to a hanging paragraph. The numbers 40, 4, 11, and 7 are tentative. We expect that the user will wish to change them. REFCOL is a control variable, set initially to 0. See Fig.4(a).

We distinguish two cases.

CASE A.

If all these conditions hold, we start a paragraph. REFCOL is set equal to the smaller of CMAR and NMAR. Besides, a control variable LEFTMAR is set equal to NMAR. The subsequent lines SUBSEQ are checked for the following condition:

SUBSEQ exists in the same page, and
either $SMAR = 0$ or $|SMAR - LEFTMAR| \leq 2$.

Here, SMAR is the number of leading spaces of SUBSEQ. Again, 99999 is assumed for an empty line. (The condition $SMAR = 0$ is a trick for saving the labor of the user when he is typing an indented paragraph. This is again tentative.)

All the subsequent lines up to the last one satisfying this condition are glued together to form a long text string containing all the material of the paragraph. (See Fig.4(b).) In gluing the lines, the leading spaces of the second and later lines in the paragraph are stripped off. Between the glued lines are inserted two spaces if the preceding line ends either with a period,

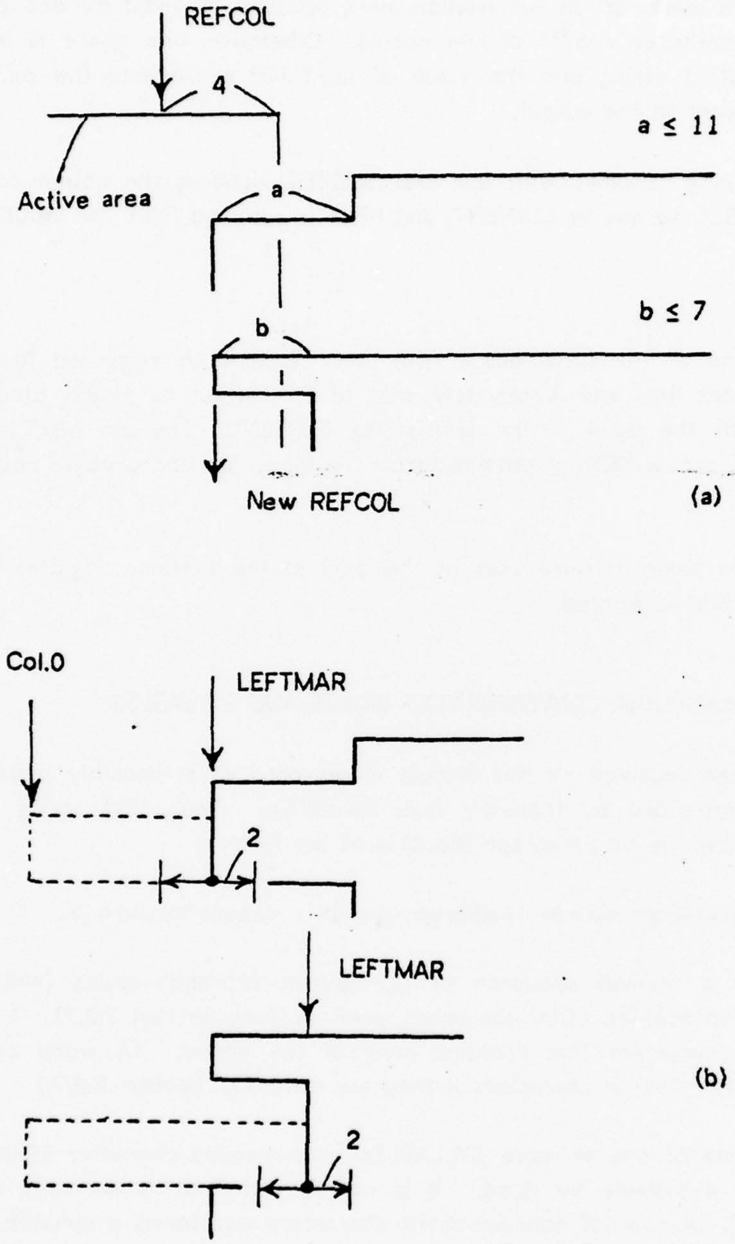


Fig.4: The discrimination algorithm for text paragraphs and independent lines: (a) Starting a new text paragraph, (b) ending a paragraph.

a question mark, or an exclamation mark possibly followed by one or more right parentheses and/or double quotes. Otherwise, one space is inserted. The resulting string and the value of LEFTMAR represents the paragraph, which is sent to the output.

The scanning resumes with the first SUBSEQ violating the above condition. This SUBSEQ serves as CURRENT, and NEXT is supplied from the input file.

CASE B.

If any one of the conditions (1)-(4) fails, CURRENT is regarded to give an independent line, and immediately sent to the output as a text block. The content of the block is the text string CURRENT. The old NEXT replaces CURRENT, a new NEXT is obtained from the input, and the process repeats.

Fig.5 illustrates some extreme uses of this part of the system. Fig.5(a) is read as input, and Fig.5(b) is printed.

2.2.3. THE ROMAN-KANA CONVERTER (1) - WORDS AND SYLLABLES

The text strings received by this module is regarded as a (possibly null) sequence of WORDs surrounded by (possibly null) PADDINGs. Thus, text string (of either independent title line or paragraph block) is of the form

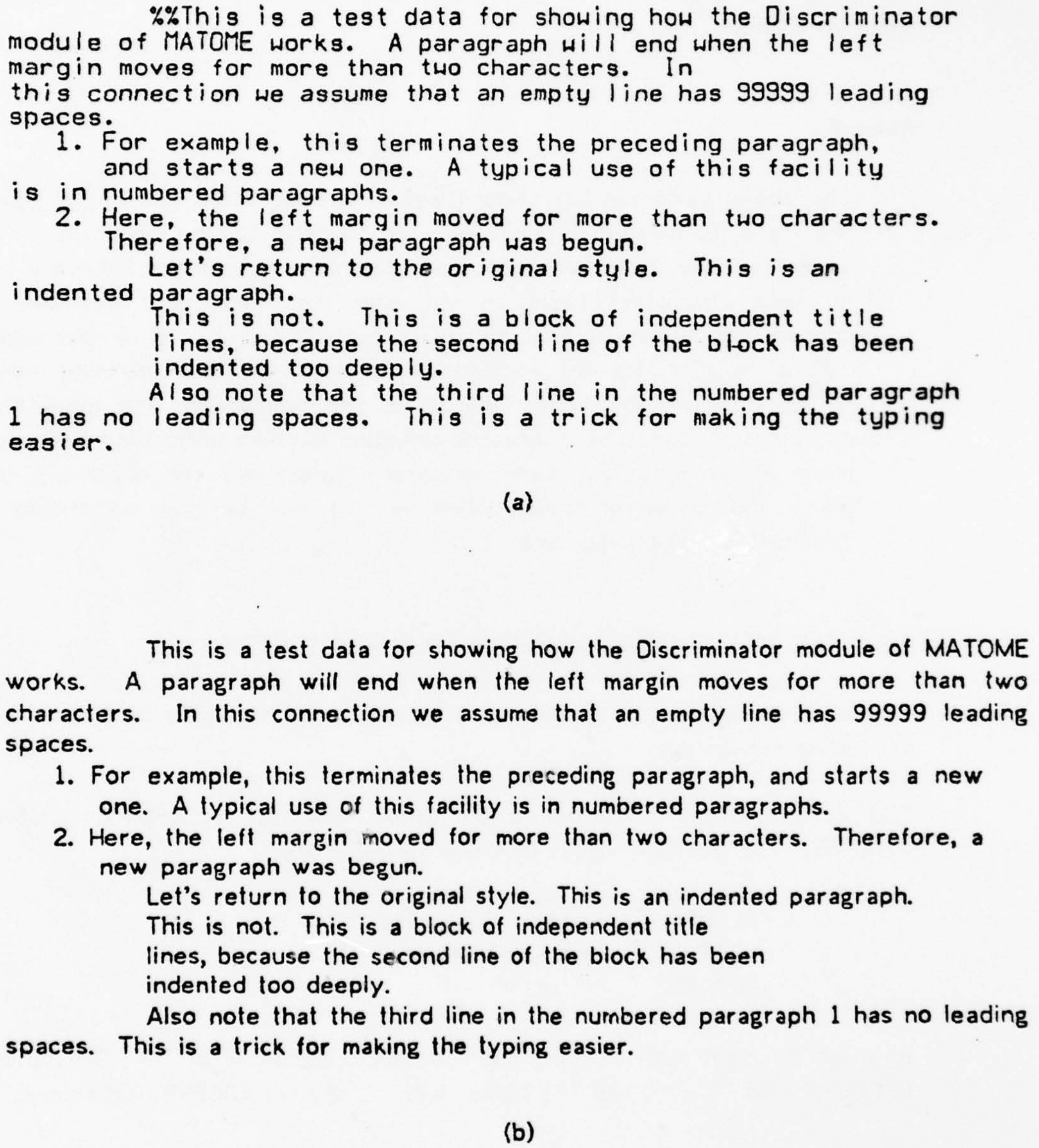
<padding> <word> <padding> <word> ... <word> <padding>

A padding is a (mixed) sequence of (1) spaces, (2) shift codes (<>%), and (3) symbolically represented LOOK commands (such as {ua}, Section 2.2.7). A word is a sequence of characters that contains none of the above. (A word can contain spaces and other special characters if they are escaped, Section 2.2.7.)

A word consists of one or more SYLLABLEs. An escaped character (Section 2.2.7) always forms a syllable by itself. It is not subjected to roman-kana conversion (Section 2.2.5). A span of non-alphabetic characters also forms a syllable. Again, it is not subjected to roman-kana conversion. A non-null span of unescaped alphabetic characters is an ALPHABETIC SYLLABLE. It is subjected to possible roman-kana conversion.

We regard double quotes, single quotes, commas, minus signs, and periods as alphabetic.

The words are converted to corresponding sequence of (variable-width) characters.



```
          %%This is a test data for showing how the Discriminator
module of MATOME works.  A paragraph will end when the left
margin moves for more than two characters.  In
this connection we assume that an empty line has 99999 leading
spaces.
   1. For example, this terminates the preceding paragraph,
      and starts a new one.  A typical use of this facility
is in numbered paragraphs.
   2. Here, the left margin moved for more than two characters.
      Therefore, a new paragraph was begun.
          Let's return to the original style.  This is an
indented paragraph.
          This is not.  This is a block of independent title
          lines, because the second line of the block has been
          indented too deeply.
          Also note that the third line in the numbered paragraph
1 has no leading spaces.  This is a trick for making the typing
easier.
```

(a)

```
          This is a test data for showing how the Discriminator module of MATOME
works.  A paragraph will end when the left margin moves for more than two
characters.  In this connection we assume that an empty line has 99999 leading
spaces.
   1. For example, this terminates the preceding paragraph, and starts a new
      one.  A typical use of this facility is in numbered paragraphs.
   2. Here, the left margin moved for more than two characters.  Therefore, a
      new paragraph was begun.
          Let's return to the original style.  This is an indented paragraph.
          This is not.  This is a block of independent title
          lines, because the second line of the block has been
          indented too deeply.
          Also note that the third line in the numbered paragraph 1 has no leading
spaces.  This is a trick for making the typing easier.
```

(b)

Fig.5: Extreme uses of the discrimination algorithm:
(a) input; (b) output.

The blanks within the paddings are likewise transformed into as many blanks, which is later expanded by the LOOK right-justification feature (cf. Section 2.2.7). Resulting sequences of words and paddings are sent to the Paragraph Cutter (Section 2.2.6).

Remark.

> The above definition has some drawback in that, if the unit of the filling and the right justification of text lines are taken to be the words in the above sense (a natural choice), then a word containing both the katakana and the hirakana characters could be cut over the lines. For example, ローマじ (romanized Japanese) may be cut into ローマ and じ. It is perhaps more natural to introduce one additional level in our definition, defining our words to be delimited by either spaces or line boundaries, and allow them to contain shift codes. Our present definition is given unmodified because much more experimentation seems necessary before we can decisively tell the gains and losses of these definitions. It may be that we should rather change the filling procedure.

2.2.4. THE ROMAN-KANA CONVERSION (2) - CASE SHIFTING

This part of the module is used as a subroutine by the main part described in the preceding subsection.

The case shifting is controlled by three variables: GLOBSHIFT, LOCSHIFT, and VERDICT. The possible values of these variables are:

GLOBSHIFT: empty, "<", ">", and "%" ;
LOCSHIFT : empty, "<", ">", and "%" ;
VERDICT : "<", or ">".

In a left to right scan of the input text, the occurrences of "<<", ">>", and "%%" set GLOBSHIFT to "<", ">", and "%", respectively. They set LOCSHIFT to empty.

Similarly, the occurrences of "<", ">", and "%" set LOCSHIFT to "<", ">", and "%", respectively. They set CURSHIFT to empty.

Just before a padding is scanned, LOCSHIFT is set to empty. GLOBSHIFT is set to empty at the beginning of a new input file.

An alphabetic syllable is regarded to be a syllable in English (no conversion) if

LOCSHIFT followed by GLOBSHIFT

starts with a "%". Otherwise, a *roman-kana conversion* (Section 2.2.5) is attempted. If the conversion is unsuccessful, then the syllable is again regarded as a syllable in English. If the conversion is successful, a "katakanization" condition (discussed shortly) is checked. If the condition holds, the VERDICT is given the value "<". If it does not, the value of VERDICT is ">". The first character in the concatenation

LOCSHIFT followed by
  GLOBSHIFT followed by VERDICT

determines the case. If it is ">", the syllable is printed in the hirakana. If it is "<", the syllable is printed in the katakana.

The katakanization condition currently used has been described In Section 3.1. In the Snobol code, it reads as follows:

```
          KATAPAT  =  ANY("'()*+0")
    +          | '3↑'
    +          | ( (POS(0) | NOTANY(']/'))
    +              ('J' | 'K' | 'L' | 'M' | 'N')  '_' )
```

From the single quote to the plus sign in the first line corresponds to the *small* "アイウエオ", "0" to the elongation sign, "3↑" to "ヴ", from "J" through "N" to "ハヒフヘホ", which are converted into "パピプペポ" by "_" (corresponding to the circle accent sign). "]/" corresponds to "n" (ん) and the small "tsu" (っ). This Snobol pattern may be changed if desired.

In the above explanation, the "occurrences" of ">", ">>", and the like more precisely mean the following. A span of "<", ">", and "%" is extracted and examined. If it is of length more than 1, and if the last two characters are equal, then these two characters are read as "<<", ">>", or "%%". Otherwise, the last character indicates "<", ">", or "%".

Remark.

> In the above, LOCSHIFT is set to empty if "<<", ">>", or "%%" is encountered. However, this is not necessary if our current definition of the word is sticked to. LOCSHIFT is set to empty anyway before one of these code combinations is processed. We require the above, and have actually coded to that effect, since we anticipate changes in the definitions as discussed in the Remark of the preceding section.

2.2.5. THE ROMAN-KANA CONVERTER (3) - THE CONVERSION ALGORITHM

This part is another subroutine for the main part (Section 2.2.3).

The conversion from the romanized form into kana characters (roman-kana conversion) is governed by a table. An edited form of the table looks as follows:

a: あ	mi: み	myu: みゅ	ja: じゃ
i: い	mu: む	myo: みょ	zya: じゃ
u: う	me: め	rya: りゃ	ju: じゅ
e: え	mo: も	ryu: りゅ	zyu: じゅ
o: お	ya: や	ryo: りょ	jo: じょ
ka: か	yu: ゆ	ga: が	zyo: じょ
ki: き	yo: よ	gi: ぎ	bya: びゃ
ku: く	ra: ら	gu: ぐ	byu: びゅ
ke: け	ri: り	ge: げ	byo: びょ
ko: こ	ru: る	go: ご	pya: ぴゃ
sa: さ	re: れ	za: ざ	pyu: ぴゅ
shi: し	ro: ろ	ji: じ	pyo: ぴょ
si: し	wa: わ	zi: じ	dzi: ぢ
su: す	kya: きゃ	zu: ず	dzu: づ
se: せ	kyu: きゅ	ze: ぜ	je: じぇ
so: そ	kyo: きょ	zo: ぞ	wi: ゐ
ta: た	sha: しゃ	da: だ	we: ゑ
chi: ち	sya: しゃ	di: でぃ	wo: を
ti: てぃ	shu: しゅ	du: どぅ	wwa: うぁ
tsu: つ	syu: しゅ	dyu: でゅ	wwi: うぃ
tu: とぅ	sho: しょ	de: で	wwe: うぇ
te: て	syo: しょ	do: ど	wwo: うぉ
to: と	cha: ちゃ	ba: ば	fa: ふぁ
na: な	tya: ちゃ	bi: び	fi: ふぃ
ni: に	chu: ちゅ	bu: ぶ	fe: ふぇ
nu: ぬ	tyu: ちゅ	be: べ	fo: ふぉ
ne: ね	cho: ちょ	bo: ぼ	va: ゔぁ
no: の	tyo: ちょ	pa: ぱ	vi: ゔぃ
ha: は	nya: にゃ	pi: ぴ	vu: ゔ
hi: ひ	nyu: にゅ	pu: ぷ	ve: ゔぇ
fu: ふ	nyo: にょ	pe: ぺ	vo: ゔぉ
hu: ふ	hya: ひゃ	po: ぽ	tsa: つぁ
he: へ	hyu: ひゅ	gya: ぎゃ	tsi: つぃ
ho: ほ	hyo: ひょ	gyu: ぎゅ	tse: つぇ
ma: ま	mya: みゃ	gyo: ぎょ	tso: つぉ

The original form as coded in Snobol 4 is shown in Section 3.1, Step 2.

In a roman-kana conversion, the given syllable is firs checked for initial occurrences of double quotes("). If there are any, they are stripped and converted into Japanese opening brackets (「). Then, the following is repeated until all characters of the syllable is stripped and converted, or a failure of the conversion is signalled.

1. The syllable is checked for an initial occurrence of one of the strings preceding colons in the above table. If there is a match, that part of the syllable is stripped, and converted into the corresponding combination of the kana characters that follows the colon. In the present implementation, uppercase and lowercase roman alphabets are assumed to have the same meanings. (However, see Section 2.3.)

2. If the above check fails, the syllable is checked against the following Snobol pattern:

```
               SOKUPAT = POS(0)  ('kk' | 'ss' | 'tt' | 'tch' | 'hh' |
+                                 'dd' | 'pp' | "t'")
```

   If this test succeeds, the first character of the syllable is stripped, and converted into the small "tsu" (っ).

3. If the above still fails, the first character of the syllable is stripped anyway. It determines the action in the following way:

   3a. Double quote (") ... If what follows within the syllable contains nothing other than double quotes, periods, and commas, a closing bracket (」) is given. Otherwise, conversion fails.

   3b. Single quote (') ... Just ignored.

   3c. Period (.) ... If what follows within the syllable contains nothing other than double quotes and commas, the Japanese circle punctuation mark (。) is given. Otherwise, a midpoint (・) is given.

   3d. Comma (,) ... If what follows within the syllable contains nothing other than double quotes and periods, a Japanese dot punctuation mark (、) is given. Otherwise, the conversion fails.

   3e. Minus sign (-) ... An elongation sign (ー) is given.

   3f. 'm' or 'n' ... Corresponding Japanese character(ん) is given.

EXAMPLES.

kana (the hirakana or the katakana) ... undergoes two applications of the rule 1, becomes かな.

"kan'a" (a crow in mid-winter, enclosed in brackets) ... after the first double quote is stripped and converted, undergoes applications of the rules 1, 3f, 3b, 1, and 3a, and becomes 「かんあ」.

ka-negi-.meron (Carnegie-Mellon) ... processed by rules 1, 3e, 1, 1, 3e, 3c, 1, 1, and 3f, and becomes かーねぎー・めろん, which is automatically katakanized into カーネギー・メロン.

gurafikkusu (graphics) ... processed by rules 1, 1, 1, 2, 1, and 1, becomes ぐらふぃっくす, and is katakanized into グラフィックス.

2.2.6. THE PARAGRAPH CUTTER

This module receives the converted text blocks from Roman-Kana Converter, and arranges them into output lines.

Let OLL (or OUTLINELENGTH) be the maximum number of XGP dots allowed in an output line. Let ILL (or INLINELENGTH) be the maximum number of characters ordinarily expected in an input line. Our standard setting is

OLL = 1098 (6 inches), and
ILL = 69.

Page marks are simply passed to the Assembler. The converted text string of a paragraph block is treated in the following way.

The first line of the output paragraph is given a leading indentation of

(CMAR' * OLL) / ILL dots,

where CMAR' represents the number of leading blanks of the text string, i.e., those of the first input line of the paragraph. We define CMAR' to be zero if the string is empty, though this never occurs here. The subsequent output lines of the paragraph are given a leading indentation of

(LEFTMAR * OLL) / ILL dots.

All the output lines within the paragraph are given a right margin at

$$OLL - ((LEFTMAR * OLL) / 2) / ILL \text{ dots.}$$

Right justification within these limits is instructed to LOOK by a combination of symbolically represented LOOK commands (Section 2.2.7). The factor 2 is also tentative. The user may wish to change it.

The filling of the lines are done by keeping track of the total widths of blank and nonblank characters. The lines are cut at the end of the last accommodatable word.

The text strings of converted independent title lines are treated as follows. First, the leading spaces are counted to give CMAR' as above. (Here, CMAR' may be zero as a result of an empty title line.) It is used to determine the amount of the indentation in the corresponding output line. Namely, a leading indentation of

$$(CMAR' * OLL) / ILL \text{ points}$$

is provided. If the line thus indented turns out to be too long for being accommodated within the output line, the rest of the material is moved to a next output line as though it is in a text paragraph with

$$LEFTMAR = CMAR',$$

but no right justification is attempted.

Remarks.

1. If the output line is so short that it cannot accommodate the first word in the converted material, the word is printed there anyway disregarding the right margin restriction. That part of the word exceeding the righthand-side limit of the paper roll will be lost.

2. In text paragraphs, it is convenient if excess spaces between the words are removed automatically. This is particularly true under automatic discrimination of text paragraphs (see Section 2.2.2 and [10]). This has not been done here because we are yet not very sure whether kana documents do not need various numbers of spaces in order to indicate grammatical relationships between adjacent words.

2.2.7. THE ASSEMBLER

An unescaped occurrence of "{" starts a command for the Assembler (Section 2.2.1, Item (d)). The command is closed by a "}", which must be in the same line. The command syntax is

```
<command> == '{' <comname>
                 [ '/' <paral> / ',' <para2> ]_ '}'
```

EXAMPLES:

{ua}	Use A-font.
{a,538}	Load font No.538 in the A-font area.
{bak,8}	Backspace 8 points.
{und/0}	Stop underlining.

The following string (extracted from the original Snobol program) gives all possible comnames. For example "eof" and "vs" are possible comnames, but "-" is not. The uppercase and lowercase characters have the same meanings in the comnames.

```
        CODENAME = 'eof vs lm tm bm lin cut nocut '
+                  'ak bk - - ua ub jw pad '
+                  's image - lf ff ecl bcl cutim '
+                  't - bjon bjoff quot ovr - - '
+                  'sup sub dcap vec sl il pag - '
+                  '- - blk und set exec bak imfl '
+                  'vcfl a b fmt rvec rvfl hnum fcnt '
+                  'break use '
```

Paral must be a nonnegative decimal integer less than 128. Para2 must be a nonnegative decimal integer less than 16384. Comname is transformed into two characters, whose respective code values equal octal 177 (escape character for LOOK) and that value given by the position in the above code table: 0 for "eof", 1 for "vs", and so on. Paral is transformed into one ASCII character with a code value as specified. Para2 becomes a binary representation packed into two 7-bit characters. For example, {a,538} becomes

octal 177 followed by
octal 61 followed by
octal 4 followed by
octal 32.

If no closing "}" exists in the same line, or if the comname is not found in the above string, or if paral or para2 exceeds the respective limit, the whole material, either up to the closing "}" or to the end of the line, is passed unmodified.

For the significance of the individual LOOK codes, see [1a].

The earlier stages ((a) through (c) of Section 2.2.1) simply pass these control codes. More precisely, that part of the input line from an unescaped "{" up to whichever occurring earlier of the corresponding "}" and an end of line is treated as a component of a padding.

Remarks.

1. This should perhaps be changed somewhat if our definition of the word is changed according to the note of Section 2.2.3. Control codes should be more transparent to the earlier stages so that it will not cause a switching from a word to a padding.

2. Although the user can write his own "{ ... }" to instruct the Assembler, the main usage of this feature is internal. The previous stages creates many of them. For example, the Japanese double-dot and the circle accent symbols are overprinted by "{bak,8}" (backspace 8 points) created within the Roman-Kana Converter (b, Section 2.2.1).

THE ESCAPE CONVENTION... The French grave accent sign (`) is used as the escape character. (Again, this can be easily changed.) It causes the immediately following character to be handled as an English syllable. Thus, "``" means the grave symbol itself, "`{" means the left brace, and so on. If the escape character is at the end of a line, it means a space.

2.2.8. THE MAIN LOOP

The man-machine interaction was outlined in Section 2.1. This subsection adds some more details. The main points are:

(1) The user of our system, if he so chooses, can edit a JPN-file, or otherwise create a new file having a name of the form "xxxxxx.jpn", and feed it to our program. If the file name is of the above form, the conceptual modules (a) through (c) are skipped, and the file is processed directly by the Assembler. If the input file name is of the form "xxxxxx.xgo", nothing occurs.

(2) If one gives an input file in his own permanent file area, and explicitly specifies the name of the area (the structure name), then the output files (JPN- and XGO-files) are created in the same permanent file area. Otherwise, the output files are created in the user's temporary file area.

More particularly, the main loop of the processing of our program proceeds as described below. (This description assumes a cursory familiarity with the file naming convention of the System-10 Monitor operating system. Those readers unfamiliar with it may wish to skip it.)

When activated, our program prompts the user as

```
          Input filename or [CR] or ? :                         (*)
```

The user types something and hits CR. Spaces are removed, and lowercase characters are converted into uppercase characters. Then, the following occurs:

1. If the material is empty, the execution of our program is terminated, and the user is taken to the SITBOL system.

2. If the material is a question mark(?), the contents of a help file is shown to the user. The system then returns to (*).

3. In other cases, the material typed is assumed to be a file name. It must specify a single file existing in the file system. The structure name and the project programmer number (PPN) is optional. If a structure name is explicitly given (e.g., as "dskb:"), and PPN is not, then the JPN- and the XGO-files are created (possibly overwriting existing ones) within that structure. Otherwise "temp:" is used as the default structure name of the output files. In particular, if a PPN is given, the input is taken as specified, and the JPN- or the XGO-files are created (or overwrite) in the user's own "temp:" structure regardless of the structure from which the input is taken. After the completion of this processing, the system returns to (*).

   If, however, the extension of the input filename is ".jpn", then the content of the file is assumed to be in a format suitable to the Assembler (Section 2.2.7). An XGO-file is created, but no JPN-file is. The system then returns to (*). If the extension of the input filename is ".xgo", the system returns to (*) immediately.

   If the syntax of the file name is incorrect, or if the input file is not found, an error message

   ```
   Usage: single-filename
   ```

   is given, and the system returns to the prompt (*).

2.2.9. ERROR MESSAGES

Only error message expected at the user terminal in addition to the one described in the preceding subsection concerning file names is the following:

```
Bad code in line xxx changed into space...
< yy .......... y >
```

xxx gives the input line count, and yy ... y displays the line in question in the original form. Our system does not expect ASCII codes 0 through 37 and 177 (in octal) within input lines (except in line terminators). These codes are changed into spaces, and the above warning is given.

In addition, an error message of the following form is possible if a help file has not been properly set up.

```
Sorry. Help information unavailable in zzzzzz
```

zzzzzz gives a file name.

The following error messages might appear only if you change the program to cause a coding conflict:

```
Warning: Empty para to CRTABLE -
      empty sequence returned...

Warning: Missing entries in CRTABLE...

Warning; Unpaired entries in CRTABLE...

Warning: ASCIISEQ includes separator as item...

Warning: ASCII sequence xxx - yyy
      requested - empty sequence returned...

...More than expected calls to GETLINE after EOF

...GETPADDING coding conflict

...GETWORD coding conflict

Unprintable character in English syllable...
```

2.2.10. CHARACTER CODES AND THE COMMAND FILE

This subject is basically an implementation issue, but it affects the user if he wishes to edit a JPN-file.

Fig.6 shows the code tables of our kana fonts. These apply to both of the fonts of height both 30 and 35. This is based on a Japanese Industrial Standard (JIS).

The canned commands assumed in Section 2.1 to be in the file "cmd[c300ik40]" instruct LOOK to load

the hirakana font of height 30 in font area 538,
the katakana font of height 30 in font area 539,
the hirakana font of height 35 in font area 540, and
the katakana font of height 35 in font area 541.

The JPN-file loads the hirakana and the katakana fonts of height 30 to the A and B-font area of the core, respectively, and suitably switches between them.

The strange numbers 538, 539, ... were chosen at random to avoid conflicts with other people's fonts.

The font switching codes {ua} and {ub} are issued only if necessary. However, the system assumes that any appearance of user-written LOOK commands may change the choice of the fonts.

Remark.

1. As discussed in Section 2.3, our mock-up have certain unimplemented features. One example is the use of the larger fonts. The fonts Nos.540 and 541, as noted above, cannot be used unless the user writes his own (symbolic) LOOK commands. This can be easily remedied, but this paper had to be written before we do so due to time constraints.

2. Since this paper includes a small number of Chinese characters, a special font was developed for this purpose. It is basically the katakana font of height 30, but a few katakana characters were replaced by Chinese characters. This feature is not described here since it is for one-time use.

Low

High	0	1	2	3	4	5	6	7
00		ゐ	ゑ	←				
01								
02	0	1	2	3	4	5	6	7
03	8	9	:	;	<	=	>	?
04	sp	。	「	」	、	・	を	ぁ
05	ぃ	ぅ	ぇ	ぉ	ゃ	ゅ	ょ	っ
06	ー	あ	い	う	え	お	か	き
07	く	け	こ	さ	し	す	せ	そ
10	た	ち	つ	て	と	な	に	ぬ
11	ね	の	は	ひ	ふ	へ	ほ	ま
12	み	む	め	も	や	ゆ	よ	ら
13	り	る	れ	ろ	わ	ん	゛	゜
14	`.	a	b	c	d	e	f	g
15	h	i	j	k	l	m	n	o
16	p	q	r	s	t	u	v	w
17	x	y	z	{	|	}	~	

(a)

Low

High	0	1	2	3	4	5	6	7
00		ヰ	ヱ	_				
01								
02	sp	!	"	#	$	%	&	'
03	(	)	*	+	,	-	.	/
04	sp	。	「	」	、	・	ヲ	ァ
05	ィ	ゥ	ェ	ォ	ャ	ュ	ョ	ッ
06	ー	ア	イ	ウ	エ	オ	カ	キ
07	ク	ケ	コ	サ	シ	ス	セ	ソ
10	タ	チ	ツ	テ	ト	ナ	ニ	ヌ
11	ネ	ノ	ハ	ヒ	フ	ヘ	ホ	マ
12	ミ	ム	メ	モ	ヤ	ユ	ヨ	ラ
13	リ	ル	レ	ロ	ワ	ン	゛	゜
14	@.	A	B	C	D	E	F	G
15	H	I	J	K	L	M	N	O
16	P	Q	R	S	T	U	V	W
17	X	Y	Z	[	\	]	↑	

(b)

Fig.6: Code tables: (a) the hirakana; (b) the katakana. The columns correspond to the lowermost octal digit. The rows correspond to two higher-order digits.

2.3. MISSING COMPONENTS

As noted previously, our program is an incomplete mock-up. Besides being big and slow, it has certain missing components. Some of these are

1. Pagination (including footnoting);
2. Centering and right-justification;
3. Switching between the fonts of heights 30 and 35;
4. Underlining, superscripting, subscripting;
5. Macro processing (including a DATE macro);
6. INCLUDE facility, especially for combining XGO- and JPN-files;
7. Facilities for changing system variables (such as output line length);
8. Refinement of the roman-kana conversion procedure;
9. Adding some error messages;
10. Expansion of tab codes into sequences of spaces.

In the present implementation, pagination(1) is relegated to LOOK. Whenever a page becomes full, LOOK puts a bottom margin, operates the paper cutter, puts a top margin, and proceeds to the next line. This is often insufficient. For example, we may wish that the titles be not divorced from the text lines immediately following them. This cannot be enforced in the present organization.

As for (2), our proportional expansion of leading spaces (Section 2.2.2) works nicely for, e.g., aligning the date with "Sincerely" in a letter, but works poorly for centered or right-justified material. Here, we would need some control codes.

For now, the user can do (3) and (4) only if he writes his own LOOK codes. These, and macro processing(5) could be done by incorporating one more preprocessing step in the sequence of Section 2.2.1.

To introduce an INCLUDE facility(6), we could slightly modify the main loop (Section 2.2.8). The design anticipates this addition. This has been left unimplemented simply because we had no time to experiment with it to determine the user interface.

One reasonable way to make the control variables changeable(6) will be to add a new option in the prompt "Input Filename or [CR] or ? : " (Section 2.2.8).

Our conversion algorithm(8) could be improved in various ways. For one thing, we could reject those syllables containing too many uppercase characters such as "USA". In the present version this becomes うさ, but we prefer it to be handled as a syllable in English. Our katakanization condition could include more. It could cover such patently foreign words as マンション (manshon -- mansion), キリスト (kirisuto -- Christ), and ゼロックス (zerokkusu -- Xerox).

Error messages(9) should be given if a very long word results in a truncation at the end of the paper roll. (See Remark 1, Section 2.2.6.) Similarly, the Discriminator should issue a warning if paragraphs are cut or continued in an unusual way. (For example, the sample text of Fig.5 should generate several warnings.)

Finally, we have another small thing (10). In the present version, a tab code in the input file, among other codes with octal code values 0 through 37 or 177, causes an error message to be issued, and is replaced by a space for further processing. We could rationalize this design by citing Kernighan and Plauger[4], and claiming that tab codes should be treated in separate filters. We could require the user to filter a file containing tabs through Kernighan's "detab" to expand the tabs into spaces. This is perhaps the only way to go if we were using UNIX. For one thing, tab codes assume a tab setting. To have separate "entab" and "detab" filters would bring more flexibility by enabling changes of tab positions. In our environment, however, our standard text editor SOS sometimes introduces tab codes automatically assuming a certain tab setting. Besides, the Monitor does not support pipelines. We therefore plan, though with a low priority, to add code to our program to expand tabs into spaces. The tab settings will be fixed to columns 0, 8, 16, ... according to the system-wide assumption of the manufacturer.

We shall return to the subject of filling these holes in Section 3.6.

Chapter 3. THE PROCESS OF THE DEVELOPMENT

In Section 3.1, we review the sequence of events which led to the present design. In Section 3.2, we try to enumerate the good things in our life-style. Section 3.3 compares it with other people's life-styles. Section 3.4 talks about pitfalls which our method could have. Sections 3.5 describes the compromising considerations done in the very early stage of the design, and finally, Section 3.6 extrapolates the chronicle of Section 3.1 to the future.

3.1. A CHRONICLE

We now briefly review what we have done. The following description has been simplified. Events occurring in parallel are described as though they occurred sequentially, and some (mostly unsuccessful) trial-and-error processes for circumventing the environmental difficulties (such as high CPU-time consumption) are not mentioned. Small adjustments (changing the format of the prompt, etc.) have been done almost daily, but these are not documented separately.

0. After a study of the environment, the vague idea of building a system for kana-English documents was reached. Ideas were also emerging slowly for *the input format and the choice of the language*. The latter ideas became clear much later.

1. A tentative hirakana font was developed with the help of BILOS[8]. An ordinary ASCII font file (called PEL25) with fixed-width characters was used as a basis. We added some blank spaces on the top of each character to increase the height from 25 to 30, and then overwrote the ASCII symbols and the uppercase roman characters with the hirakana characters copying that portion of the standard code table given by a Japanese Industrial Standard (JIS). The lowercase roman characters remained in the standard position. A sample text file was printed on the XGP. Since only one font was involved, and since the hirakana characters replaced ASCII printable codes, the sample text could be readily created by hand. All the possible combinations of the characters with the accent signs were also included. In this step, accent signs were printed separately without overprinting. They occupied some small width.

   We note that our code table (Fig.6) was a result of a second thought. First we tried another code table induced by standard kana keyboard (for which there was another JIS). For example, we replaced "q" with "ta" (た) because in the kana keyboard "ta" was the leftmost key on the second row. We soon discarded this idea because the combination was too capricious to memorize, and we had to overwrite randomly chosen uppercase and lowercase characters. (For a related subject, see Section 3.5.2.)

This font was periodically reviewed and retouched. We will not give the details of the process here. One major change was that the font PEL25 used as a basis was later replaced by more standard variable-width font called NGR25.

2. A syllable table of the following form was produced. (This is the original form of the table discussed in Section 2.2.5.)

```
        TABLE = 'a i u e o ka ki ku ke ko sa shi si su se so '
    +           'ta chi ti tsu tu te to na ni nu ne no '
    +           'ha hi fu hu he ho ma mi mu me mo ya yu yo '
    +           'ra ri ru re ro wa '
    +           'kya kyu kyo sha sya shu syu sho syo '
    +           'cha tya chu tyu cho tyo nya nyu nyo '
    +           'hya hyu hyo mya myu myo rya ryu ryo '
    +           'ga gi gu ge go za ji zi zu ze zo '
    +           'da di du dyu de do ba bi bu be bo pa pi pu pe po '
    +           'gya gyu gyo ja zya ju zyu jo zyo '
    +           'bya byu byo pya pyu pyo '
    +           'dzi dzu je wi we wo wwa wwi wwe wwo '
    +           'fa fi fe fo va vi vu ve vo tsa tsi tse tso '
        ENTRY = '1 2 3 4 5 6 7 8 9 : ; < < = > ? '
    +           '@ A C( B D) C D E F G H I '
    +           'J K L L M N O P Q R S T U V '
    +           'W X Y Z  [ \ '
    +           '7, 7- 7. <, <, <- <- <. <. '
    +           'A, A, A- A- A. A. F, F- F. '
    +           'K, K- K. P, P- P. X, X- X. '
    +           '6↑ 7↑ 8↑ 9↑ :↑ ;↑ <↑ <↑ =↑ >↑ ?↑ '
    +           '@↑ C↑( D↑) C↑- C↑ D↑ J↑ K↑ L↑ M↑ N↑ J_ K_ L_ M_ N_ '
    +           '7↑, 7↑- 7↑. <↑, <↑, <↑- <↑- <↑. <↑. '
    +           'K↑, K↑- K↑. K_, K_- K_. '
    +           'A↑ B↑ <↑* ' ASCII(1) ' ' ASCII(2)
    +                                   ' & ' "3' 3( 3* 3+ "
    +           "L' L( L* L+ 3↑' 3↑( 3↑ 3↑* 3↑+ B' B( B* B+ "
```

For example, "ge" in the eighth line of TABLE corresponds to "9↑" of the eighth line of ENTRY. The ASCII code position for "9" is in fact occupied by the syllabic character "ke" (け), and that for "↑" is occupied by the double dot accent (dakuten), which converts "ke" into "ge" (げ). The table was proofread by writing a simple Snobol program, which generated something like the edited form of Section 2.2.5. This could be done easily because the lowercase roman alphabet remained in the font.

In fact, the above table includes later additions. (The same applies to other tables shown in this section.) For example, two characters used only in Japanese classics were added and corresponded to ASCII code positions 1 and 2, since we wished to take some of our sample texts from celebrated sources. The calls to the SITBOL built-in function ASCII in the above reflect this fact.

3. A corresponding katakana font (of height 30) was created and tested using the same set of test data as in Steps 1 and 2. The code tables had the identical shapes. (E.g., both the hirakana and the katakana forms (け and ケ) of the syllable "ke" replaced "9".) This contributed considerably to the ease of testing. The same test data could be used unmodified.

   It is interesting to note that, while the hirakana font required several sittings before it reached a tolerable level, the katakana font reached about the same level in three or four hours. The katakana is a much easier font to develop.

4. Now a strong incentive was felt toward printing a sample text containing both the hirakana and the katakana. The test data of Step 1 actually contained some imported Western words. Attempts were made in vain to include necessary LOOK control codes by the standard text editor SOS. A few sittings were wasted exploring the properties of the environment.

   Actually, this was the most frustrating part of the project. Some of the code combinations required by LOOK (Section 1.6) were in fact the worst conceivable ones. The text editor SOS allowed us to manually construct a code combination that instructed LOOK to switch to font A, but unfortunately, we could not first go to font B. To do so, it was necessary to give a code combination

   octal 177 followed by octal 15.

   But octal 15 happened to be the carriage return. SOS defied the author's efforts to produce this code combination. As soon as it was given a carriage return code it added a line feed code to finish a line.

   This difficulty could have been solved in several ways. For example, we could write a small program in the assembler language (of PDP-10), or in some other system implementation language such as Bliss or SAIL, and try to link it to the rest of the program. In view of the nature of the processing expected, however, the best way to proceed seemed to use a local Snobol dialect SITBOL, which had a special feature for writing binary files of the type we desired. This resulted in a slow system, but the penalty seemes to be cheap enough. Using Snobol, we could smoothly package the whole processing.

5. It was now decided to write an "assembler" for converting symbolically represented LOOK codes into binary forms. Thus, if we face a naked computer, the first thing we would do is to write an assembler in order to hide some of the dirtiest aspects of the machine from the eyes of ourselves. Here we were facing a dirty interface of a (fancy) printer. Hence, we wrote an "assembler", i.e., a simple program that enables one to write LOOK control codes in a symbolic notation, and translates them into funny code combinations required by LOOK. First of all, a table of mnemonic codes were created as a long string maned COMNAME, as shown in Section 2.2.7.

   The "mnemonic codes" were first developed from the prose description of [1], but afterwards we learned that a program called TYPER written by Joe Newcomer was available for the reverse conversion (printing a symbolic representation of LOOK-oriented file). The code table was changed to reflect the TYPER names of the codes.

   This process required several sittings because some typos in the Snobol program were masked from the author's eyes by the mental pressure of attempting an unusual use of the SITBOL system.

   The test data of Step 1 augmented with symbolic control codes {ua} and {ub} finally gave a mixed printing of the hirakana and the katakana.

   It is important to note that, although at this time we needed only a few of these mnemonics, we included all of them. To do what can be done now often smoothes our later efforts.

6. In parallel to Step 5, the table of Step 2 was used for writing a separate conversion program. In the first step, the shifting between the hirakana and the katakana was simply not done, and improbable appearance of a consonant in a romanized text was converted invariably to the small "tsu" (っ), because it often gave the right answer. In short, the easiest handling of the exceptional cases were made for minimizing the investment of the labor needed to give the first visible result. The conversion routine was tested with a hirakana-only text (a classic poem). No filling of the lines was done, but this did no harm (except that the result was less beautiful). For, a roman kana conversion always caused the lines to shrink.

7. The shift codes (for forcing the katakana or the hirakana) were now introduced into the program of Step 6. We planned to provide both local

and global shifts, but here we restricted ourselves to the latter. Actually, the codes "<" and ">" were translated directly into "{ub}" and "{ua}", which instructed LOOK to switch to the A- and the B-fonts. The first section of the Appendix was now written and used for a test.

Note that the material includes roman alphabetic characters and other ASCII symbols. At this stage, they were printed in the hirakana as nonsense phrases. This gave a strong incentive for improvement.

8. The programs of Steps 5 and 7 were now combined to enable a conversion by one stroke. However, the intermediate result (the output of Step 7) was also made available for inspection. Our JPN- and XGO-files thus emerged.

9. The program now looked rather lengthy and complex. (It now occupied three line printer pages.) Therefore, a clean-up was done. Functions were defined for each of the data converting steps, and were given filenames as parameters. This step, though added nothing new, made the program much easier to understand, and at the same time inoperative! The reason later turned out to be that, if we type

```
DEFINE('JPNXGO(JPNFILE, XGOFILE)')
```

instead of

```
DEFINE('JPNXGO(JPNFILE,XGOFILE)')
```

the SITBOL system says nothing at the compile time, and does some mysterious things during the interpretation. This caused several hours to be wasted.

10. Meanwhile, the hirakana and the katakana fonts of height 35 were developed. The motivation came from the frustration while trying to push the curved strokes of the hirakana into the limited number of dots available in a character in the smaller font (of height 30). Here, the size adjustment command of BILOS helped greately.

11. In the simplified form of the "assembler" as implemented in Step 5, no facilities were provided for attaching numerical parameters to the LOOK control codes. The codes for switching between the A- and B-fonts don't require such numerical parameters, but those for instructing to swap the fonts must be accompanied with a numerical parameter designating the font serial number within the disk of the PDP-11. The use of parameters were now supported by extending the grammar of the symbolic control codes. Our use of slashes and commas thus emerged.

An attempt was now made to make a mixed use of the fonts of the heights 30 and 35. The system, however, rejected the effort giving a mysterious reason: "CORE FULL". (A trick to circumvent this difficulty was discovered later.)

12. The lock-unlock (global/local) mechanism for the shift codes "<", ">", and "%" was now implemented.

13. An opening sequence of LOOK commands was now introduced which automatically loaded the A- and the B-fonts from the disk. (Prior to this, the fonts had to be explicitly loaded to the PDP-11's core by LOOK commands.)

14. The roman alphabets and the ASCII symbols were now made printable. The symbols (including the digits) were pushed into unused positions of the code table. This slightly violated our philosophy of making things visible. (See 3, Section 3.2.) Automatic discrimination of the katakana syllables (katakanization) was also supported here. (As discussed later, this chunk was definitely too large. It caused many troubles.)

15. As the program grew bigger, the system grew slower and slower. Even the initial compilation and the definition of functions now spent considerable time. To ease experimentation, we now changed the program so as to process two or more files within the same Snobol run. (The clean-up of Step 9 was done partly because this was anticipated.)

16. The symbol "{", which the Assembler always interpreted as the beginning of a control code, was now made printable in terms of an escape convention. This change affected both the Assembler and the Roman-Kana Converter.

17. The overprinting of the double-dot and the circle accents was now implemented. This was first tested by writing a simple Snobol program, which converted an intermediate output (JPN-file) into a form including LOOK commands for overprinting, and by processing the result by the Assembler to form a LOOK-readable file. After we were sure that the result was pleasing to the eye, this change was incorporated in the program.

18. The katakanization algorithm was now slightly improved. The user was now allowed to drop the shift code "<" in such words as "puroguramu" for "program".

19. By this time, the program became painfully slow. To calm the user down, he was now given an indication of the progress by the "*"'s on the terminal. This was done by first defining an encapsulated input module, and then

modifying the result so as to count the lines behind the scenes. We discuss this in Section 3.2.1.

20. Until this time, the assembler was using the "U"-mode ("undefined" mode) of SITBOL input/output[9]. It was discovered that the simpler "T"-mode ("teletype" mode) worked equally well. The system was changed accordingly.

21. As a preparation for right-justification of the text, a set of tables of the character widths was introduced into the program. The tables were prepared for each of the hirakana and the katakana fonts of the heights 30 and 35. They were extracted directly from the font files by means of BILOS and a Snobol program. The table for the hirakana font of the height 30 looked like

```
           W.HIRA30  = '- 20 20 16 - - - - - - - - - - - - - '
    +        '16 16 16 16 16 16 16 16 16 16 7 6 14 16 14 11 '
    +        '12 12 20 20 12 20 20 18 18 18 18 18 18 18 18 18 '
    +        '20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 '
    +        '20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 '
    +        '20 20 20 20 20 20 20 20 20 20 20 20 20 20 0 0 '
    +        '7 14 16 13 15 15 10 15 15 7 9 14 7 20 15 15 '
    +        '16 15 12 13 10 15 15 21 14 16 14 11 7 11 16 '
```

Corresponding TABLEs (Snobol data objects) were created within the core, but no further processing was done at this step. (Surprisingly, this caused the later stages of the program to run almost twice as slower than before. For a discussion, see Section 3.3.3.)

22. A piece of code was now added for counting the total widths of the words, and writing the results on a file. The results were inspected. The widths were indeed calculated correctly.

23. Further pieces of code were added so that, if the total width of the (converted) characters within a line exceeded a certain limit given by a symbolic constant, the line was cut apart at the end of that word, of which the next word caused an overflow. This was tested by first creating a file containing very long lines by hand. A second test was then done by writing a separate Snobol program which automatically pasted the lines of a text paragraph to form a very long output line.

24. The separate Snobol program of Step 23 was incorporated into the system. The leading spaces of the lines were now translated into suitable tabulation codes for LOOK.

25. The discrimination technique for text paragraphs generally followed [10]. The more sophisticated method described in Section 2.2.2 was now introduced.

26. Those paragraphs having indented left margins were now made to have indented right margins, too.

27. Page marks in the input file were neglected up to this point. They were now supported as described in Section 2.2.2. (This required considerable experimentation on the SITBOL input/output.)

The work (excluding Step 0) began in late December of 1977. Most of the program writing was done in March and April of 1978. We began writing this paper in early April. May and June were spent mostly for the paper, though programming activity was still going on slowly. Due to the author's absence from CMU, three weeks and one week were lost from mid-January to early February and in May, respectively. In the average, the author worked half-time. The effective span of the development excluding documentation was about four months (from January to May, subtracting four weeks).

Step 1 through 5 were done in December, January, and February. Steps 6 through 16 was done in March, Steps 17 through 22 in April, Steps 23 through 25 in May, and Steps 26 and 27 in June.

Although the exact man-hours have not been recorded, a crude estimate might be one half of 200 hours per month for four months, i.e., 400 hours. This includes clerical tasks such as text editing, waiting for an access to the XGP in front of a terminal, and so on. Very often one activity addressed different facets simultaneously, but we would venture to say that 100-150 hours were spent over the development of the fonts, and 100 hours for studying the properties of the environment. The rest (150-200 hours) was spent for thinking about the user interface and for actually writing the program. The latter two activities are hard to distinguish because they interacted so closely.

Could we finish this work within two months if we did it as a full-time job? Perhaps not. For one thing, thinking about the user interface required some time span. We often picked up good ideas over a cup of coffee, scanning new issues of journals, and so on.

Was it a fun? Yes, very. Why was it a fun? Perhaps because almost always we felt we were progressing steadily. We knew how to proceed even when we got stuck.

3.2. ANALYSIS OF OUR LIFE-STYLE

We shall now try to see why we could proceed so smoothly.

One basic fact is that our program is small. It has only about 600 Snobol statements even in the final form. It occupies only about 25 pages in a line-printer listing. Moreover, about 130 out of the 600 are labels written as independent statements. The choice of the programming language played an important role.

Another basic fact is that we worked in a fancy interactive environment. Except for the accesses to the XGP and the graphic displays, computing resources were more than abundant. Minor changes and experiments were quite easy. If we were working in a card-based batch environment, we would have to proceed differently.

Programming languages and operating systems are most commonly imposed on the programmer. It is not always meaningful to talk about the choice of them. We shall set them aside for now, and try to enumerate other factors which possibly contributed to our success. Some of the following points are closely related to the works of Sandewall, Kernighan and Planger, and others. We shall discuss about these works in Section 3.3. We have eleven items.

1. SMALL CHUNKS FOR USABLE HALF-PRODUCTS... The development was done in small chunks, so small that one chunk usually required only one sitting. The half product was actually used for practical purposes. For example, it was used for writing the shortened Japanese version of this paper given in the Appendix. This provided nice motivation, and enabled us to detect blunders early.

2. USE OF TABLES... Whenever possible, a table was used. We attempted to make our tables as commonplace as possible. For example, the code table of TYPER was copied almost in verbatim (Step 5, Section 3.1). The machine time spent for converting these tables into more easily processed forms was just ignored. (The subject becomes of a real concern only after we complete our mock-up, collect experiences with it, and become so sure about the validity of our design that we decide to build a commercial product. We might then use, e.g., macros for preprocessing the tables.)

3. MAKING THINGS VISIBLE... The concepts were given printed forms. We tried to make everything visible. Sometimes we did need invisible things, but, even then, we built corresponding monitor apparatus (routines) as a part of the initial design. (However, we should not spend too much time over the construction of monitors. We should use off-the-shelf monitors such as a manufacturer-supplied dump routine whenever available. In our case, TYPER was used as one of the monitor apparatus.)

4. PURITY OF THE PHASES... Each of the components of our data transforming chain was made to do just what was necessary to do. For example, one phase passed commands for later phases unprocessed. This helped in experimentation.

5. WRITING PROGRAMS TWICE... For major chunks, we wrote the program twice. Thus, we first jotted down a brief description of what we were to build, possibly with diagrams. We then wrote a corresponding program. We tried to relax. We didn't strain to save paper. Once the piece of program was completed, we rewrote it carefully and neatly. (Very often we discovered an oversight in the process.) Text editing of the program file was done over a listing marked with corrections and carrying indications of the location for inserting the new material. Free-hand changes were avoided. We sometimes used two copies of listings. The corrections were first written into one of them, and then copied neatly onto another. The editing was done with the (second) copy at hand. The verbal description (with diagrams), the rewritten copy of the program, and the second listing were kept for the record. The first copies were discarded. (It must be discarded immediately in order not to mess up your desk.)

6. COOLING PERIOD... For major chunks, we introduced a cooling period. Corrected files were not run immediately. Another listing was taken, and reread. Very often further corrections were found necessary at this stage. Here, corrections were made at leisure. The listings used were kept only for two or three weeks, and then discarded. If we were not sure about the correctness of our text editing, we often waited until next morning.

7. CLEAN-UPS... As soon as the program became lengthy and complicated, a clean-up was performed. For example, we collected data elements, and encapsulated them into a collection of (Snobol) functions. If necessary, the order of the function definitions was modified. The names of the variables were adjusted. Comments were added. This was done PERIODICALLY. In a production programming environment, the preferred practice is to formulate a coding standard in advance. However, such a disciplined practice often interferes with flexibility necessary in the development of a mock-up.

   A more detailed illustration of a clean-up is given in Section 3.2.1.

   These clean-ups tended to give poor results if done simultaneously with other enhancements.

8. REVIEWING THE HISTORY... As is customary, a history of the development was kept as a comment at the beginning of the program, and was updated on the

spot. We made a casual scan of these records periodically in order to make it sure that we were on the right track.

9. SHOWING TO OTHERS... The intermediate results were shown to any willing colleagues, and their opinions were solicited. One good point of our casual way of life is that the half product can be shown to people in a visible form. This helps to get good opinions. Moreover, it is well-known that by trying to explain to others we tend to discover our own oversight, even if the audience does not understand at all!

10. WISE RETREAT... We stopped early. We knew that we were developing a mock-up. Our object was the design, not the program itself. Just as safety in moutaineering depends on wise retreat, our success required that ambition be controlled. We reviewed while we had enough time, and before we got tired. For one thing, we wrote this paper! If an incentive for improvement is felt at this stage, we should better redesign a new program, using the old one as a pattern.

11. HEALTH CONTROL... We were careful about our health, though this may sound irrelevant. We tried to keep regular time. Whenever we felt tired, we didn't hesitate to take a rest. It is of no use to be in a hurry when our health declines. In such cases, we should limit ourselves to a mechanical clean-up or a review, at the most. (Even these should better be postponed.)

3.2.1. AN ILLUSTRATION OF CLEAN-UP

For a simple example, consider Step 19, Section 3.1. In the originally written code, the initial input (text file) was read by in-line reference to an input-associated variable. Thus, we first executed the SITBOL library call

```
INPUT('ROM', INFILENAME)
```

and then referenced the variable ROM directly.

The variable ROM was now hidden in a set of Snobol functions. Two functions were defined: INITGETR(FILENAME) for executing the INPUT library call; and GETR() for getting a line from the input file by referencing ROM as follows:

```
*
*...... MODULE: GET ROMANIZED JAPANESE ......
*
        DEFINE('GETR()')
                                                :(ENDGETR).
GETR
        GETR  =  ROM                   :F(FRETURN)
                                                :(RETURN)
ENDGETR
*
        DEFINE('INITGETR(INFILENAME)')
                                                :(ENDINITGETR)
INITGETR
        INPUT('ROM', INFILENAME)
                                                :(RETURN)
ENDGETR
```

The readers familiar with Snobol 4 will see how this crazy way of coding helps later changes. After this clean-up, it was a simple matter to add code to GETR for counting the number of lines and writing #'s for each ten lines. The count was initialized in INITGETR. This sequence of events is typical in what we have done. In this way, we introduce some amount of inefficiency, but it does not really matter in a mock-up.

We note that this portion of the program was further changed. For example, in Step 24, GETR was made to return amalgamated paragraph lines, and to INITGETR was added certain code for doing some more initialization.

3.3. DISCUSSION

3.3.1. THE CONTROLLED SLOPPINESS

Our "sloppiness" has undoubtedly shocked some readers. They might object that what we have done is nothing but the long-despised bottom-up design. We now try to defend our way of life. Our thesis is that sloppiness is essential in the conception stage, and that our sloppiness is in fact controlled.

People often talk about the need of precisely formalized specifications. Many things has been said about the form which these specifications ought to take, and thier possible uses in the development of large software. Little has been said, however, about how we can get to them. Before we can formalize, we must have a clear picture of what is formalized. The mental process involved has not been analyzed fully.

What we did here is a case study for exploring this mental process. We were guided by our own intuition. People say that a program must be preceded by a formal specification. This is a reasonable requirement for a marketed program product. In our case, however, a program must precede a specification. We program for developing the latter. Our program is a mock-up, to be thrown away once a satisfactory formal specification is obtained.

A review of the steps of Section 3.1 reveals that the succession of events has been governed by logical necessity. For example, we could not determine whether the fonts of height 30 resulted in enough legibility until we experimented with it. This knowledge, however, could influence the program structure. If we got a negative result in this respect, the basic approach in the use of the XGP would have been different. We therefore developed the fonts first, and tested them. To test the fonts we needed sample data. Therefore we wrote a converter (Step 6, Section 3.1) before we began to design other parts of the program.

As noted in Section 2.2.3, our previous definition of words later turned out to be defective. This, however, was not until we came to the subject of line filling and right justification. In fact, more accurate statement is that the definition turned out to be incompatible with the later extension. In a moralistic view of programming, the author may be accused for the sin of sloppy near-sightedness, but he is not ashamed at all.

While we are addressing a rudimentary part of the system, we should not be distracted by more advanced topics. When we developed the notation for case shifting, our central concern was to reach a reasonably uniform notation comfortable for the user. The question of line filling was just ignored. We arbitrarily picked one possible definition of words, experimented with it, were generally satisfied with it, and decided to proceed.

The author feels that there is nothing wrong with this approach, even though the particular definition did not fill the needs of one of the later stages of the design. We cannot address many things at the same time. We must proceed step by step. We must pick a solution anyway if we are to go one step forward. The ultimate inadequacy of the particular solution does not matter. It is more important that the solution is formulated in a flexible way, so that we can smoothly modify it later.

Could we avoid the later modifications by thinking about line filling in the first place? Yes, of course we could. Doing so, however, would have traded real experimentation for a mental exercise. In this particular problem, this would have done no great harm, but in general, experimentation cannot be dispensed with without losing something.

3.3.2. RELATION TO OTHER WORKS

It is not surprising that the Lisp users' life-style recently described by Sandewall[3] is quite similar to ours. They program for unknown problems. Sandewall gives a small sample showing how a typical Lisp user might develop software. A program, which finds suitable meeting hours from a list of schedules of people, and updates the schedules, is described. The method, called STRUCTURED GROWTH by Sandewall, starts from a very modest fragments of the program the developer has vaguely in mind, and extends them by an alternation of writing a small increment and testing it by actually running it for a small sample data. The sample data is a part of what is developed by structured growth.

Our use of small chunks for usable half-products amounts roughly to the same thing. Our policy of making things visible also has a counterpart in Sandewall's description. He claims that the Lisp programming system is helpful to the user because every piece of data or code is an S-expression, and hence printable as such.

However, there are differences. In Sandewall's case, the product is an object in itself. Unlike in our case, he need not extract a specification from the program. This difference makes his method insufficient for our purposes. We must crystallize our ideas in the process of developing a program. We need more.

It is nevertheless reassuring that the life-style of experienced researchers in doing research-oriented programming partly coincides with ours. A similar line of thought is found in Teitleman[12].

Another related idea is found in left-corner construction philosophy of Kernighan and Planger[4]. They develop software involving man-machine interaction in chunks. For example, a file archiving program is developed in this way. The idea is "to nibble off a small manageable corner of the program -- a part that does something useful -- and make that work. Once it does, more and more pieces are added until the whole thing is done" (page 86, [4]). This is exactly what we have done (in 1, Section 3.2).

The use of tables (2, Section 3.2) has been recommended by Kernighan and Plauger[13]. The purity of the phases(4) has been practiced by Kernighan and Cherry[14]. Writing programs twice(5) and having a cooling period(6) are corollaries of the author's previous thoughts[15]. Showing to the others(9) has been advocated by Wada[16]. Perhaps new is the explicit mention of clean-ups(7), reviewing the history(8), wise reteat(10), and health control(12). We feel that the most important among them is clean-ups.

3.4. PITFALLS

Our project was a fun for the most part, but in a few places it was not. Whenever the author lost control of himself, the process became frustration. The worst of such events occurred when the author tried to do the development in the usual pace under the mental pressure of trying an unusual use of the SITBOL system (Section 3.4.1), and when he impatiently tried to do a big chunk (Section 3.4.2).

3.4.1. UNUSUAL USE OF THE PROGRAMMING SYSTEM

The first loss of control was experienced in Step 5, Section 3.1, when we tried to write an "assembler" using the "U"-format input/output of SITBOL. This format allowed us to write machine words onto an external medium. The step aimed at developing a stand-alone program, which read a text containing symbolic representations of LOOK commands, and converted it to a corresponding file suitable for sending to LOOK. Here, we concentrated on the switching between already loaded in-core fonts A and B. It was not necessary here to handle parameters in LOOK commands.

As mentioned earlier, the listings and other working documents excepting temporary memos were kept throughout this project. This Step generated seven listings. The author really hates to say this, but an analysis of these reveals that he was singularly amateurish in this particular Step.

During the process we had to address the following tasks:

1. To find how a binary file containing LOOK commands can be written by a SITBOL program;
2. To determine the command names for symbolically representing LOOK commands;
3. To determine the format of the symbolic commands;
4. To implement the code table for the "assembler";
5. To decide what to do if a wrong mnemonic code is given; and

6. To develop the man-machine interface for the user to activate the program.

Tasks 2 and 6 were handled successfully. As for the former, the mnemonic codes were changed once, from the author's own version to an in-verbatim copy of TYPER codes. This was done smoothly because we were using a table. As for the latter, the first version did not allow the user to type file names in the lowercase. This restriction was later relaxed. A facility to output help messages was added. In the initial form, the prompt was followed by a New Line. This was corrected. The user's response was made to appear in the same line as the prompt. All these went smoothly in a controlled, conventional way.

Task 4 went somewhat less smoothly. At first, the table was kept in dynamically created names. We used, in the Snobol notation, $('@' COMNAME) and the like. We then somehow felt that the practice was distasteful, and decided to use Snobol TABLEs instead. Perhaps this should have been postponed, since the subject was basically an implementation issue.

Less smooth but still satisfactory was Task 3. Here, we changed the format several times. The changes accounted for our increased awareness of the necessity to incorporate parameters later.

In the initial attempt for Task 5, we just ignored the wrong codes, and sent them to the output as though they were portions of the text. We made bad typos here, and could not find them soon. Tasks 1 and 5 impeded each other.

In Task 1, we first tried the default mode of the SITBOL output. This caused some parts of the output lines to be mysteriously lost. We then learned about the "U"-format, tried it, and was griped by the system for committing the crime of overflow. A sequence of guesses based on rather cursory description in [9] made us gradually approach the right solution, but a few sittings were wasted before we fixed the typos relating to Task 5. We were not completely sure. We changed impertinent parts of the program. We were distracted by the mental pressure.

The moral is clear. Don't do other things when you are experimenting with an unknown programming system. Proceed conservatively when you are doing a dangerous job.

3.4.2. DOING TOO MUCH AT A TIME

Another confession relates to Step 14, Section 3.1. When our system succeeded in producing a mixture of the hirakana and the katakana, the author felt a very strong incentive toward making roman alphabets and digits printable (especially the latter).

He became impatient looking at "だい1しょう" (Chapter 1) printed as "だいあしょう" (Chapter あ), and "CMU の XGP" (the XGP of CMU) as "っむ の っっっ" (a nonsense phrase). The author could not resist the temptation to sit down for three times longer than the usual, and design and implement all of the related part of the system. He rewrote the program three times, and very carefully proofread each time. Nevertheless, this gave rise to bugs, which lasted more than one day. All were typical careless errors, such as leaving out POS(0) when a pattern was to be stripped from the beginning of a string. The process involved a pain, rather than a fun. It is frustrating to end one day's work when you know that there is a remaining bug.

3.5. DISCUSSION OF STEP 0

Some more remarks are in order concerning Step 0 of Section 3.1. The question is: how did we choose the particular combination of (1) the input format, (2) the processing program, and (3) the output format? This involved a complicated set of compromises. We now sketch some of our thoughts. We begin with (3), proceed to (2), and then come to (1) since the earlier topics in this sequence had decisive influences over the later ones.

3.5.1. THE OUTPUT FORMAT

Basically, our problem calls for doing some nice things using the XGP in the production of Japanese documents. With this broadness, our thoughts resembled a negotiation between a prospective customer and a developer.

The customer desires a fully general system handling the kanji. Indeed, the XGP itself, if properly controlled, can print any graphic pattern. He knows this from, say, [1], points out that the kanji characters are special cases of graphic patterns, and demands that the possibility be pursued.

However, the implementor knows that this is infeasible. The mode of usage of LOOK explained in Section 1.6 is called the text mode. In it, a pattern corresponding to a given character is picked from one of the in-core fonts A and B. This achieves a great compression of information.

Unfortunately, we have more than 2,000 kanji characters, and the in-core fonts are restricted to at most 254 kinds of characters. If we are to print the kanji, we cannot use the text mode. Instead, we must prepare a huge bit table (image file) representing the completed document as a collection of dots. This must be created on the PDP-10, and transmitted to the PDP-11. The size of data we must handle would be at least ten times larger. This would necessarily result in cumbersome handling and slow processing. Besides, the development of the kanji font alone would require a man-year effort, and, as discussed in Section 1.5.5, the development of input technique for the standard Japanese documents is one of the unsettled problems of the Japanese text processing. The customer, no matter how ambitious he may be, will eventually agree with the implementor that the general approach should be dropped for now.

The developer, on the other hand, might propose a cheap solution based on romanized Japanese. In this solution, he has many things already available in the computer system: text editors, formatters such as PUB[6], a large collection of fancy font files, and so on. Basically, his job would be to enhance the environment. For example, he could write a simple program for editing the input in order to facilitate overscoring used in the standard systems of romanization for representing elongated vowels. He could also tailor the text editor in such a way that the changes can be made for syllables rather than individual characters. He might write a utility program for detecting and correcting impossible combinations of characters in the input file.

However, the customer would undoubtedly feel that this solution is unduly cheap, and does not deserve any investment. It is all too clear that the resulting system cannot be used for writing a scientific paper or a letter. The parties would eventually reach the agreement that our kana-English approach be pursued. The customer will probably insist that the decision be tentative, and that after a more feasibility study, the project may be dropped.

3.5.2. THE INPUT FORMAT

One obvious solution from the customer's viewpoint is to base our representation on one of the established systems of romanization. We do not have a katakana keyboard in the particular environment. We must use an ordinary ASCII keyboard. Under these circumstances, this solution is certainly natural.

Unfortunately, this "obvious" solution was not at all obvious for the implementor. First, the standard methods of romanization are such that no mechanical procedure for transliteration into a kana representation is possible unless we go into the meanings. For example, the case-designating particle "ha" which (usually) follows the subject of a sentence is pronounced "wa", and therefore written "wa" in the standard systems. But It is definitely "ha" (は) in the kana notation. Can we automatically change "wa" into "ha"? No, because the syllable "wa" (わ) as a noun means a ring! Roman-kana conversion taken literally is one of the very difficult unsolved problems in Japanese text processing essentially equivalent to kana-to-kanji conversion (Section 1.5.5). Hence, our own system of romanization.

There is another, entirely different approach: we could ask the user to use the ordinary ASCII keyboard as though it were a kana keyboard. Thus, he is to type "q" if he actually wishes to type "ta" (た). He types "p" for a double-dot accent (dakuten). He types "c" for "tsu" (つ), and "C" for small "tsu" (っ). One big merit of this method is that, if you are a perfect touch typist of the katakana keyboard, you can save almost one half of your keystrokes.

The second method, however, has the drawback that it is almost impossible to proofread the file without printing it on the XGP. The correspondence has been governed by historical reasons, and is almost as capricious as "qwertyuiop".

If this project had been done in Japan, this second method would have been the first choice, since character displays and typewriters in Japan often includes katakana characters. (This sacrifices the lowercase alphabet.) Since we were in CMU, this was simply impossible.

3.5.3. THE PROCESSING PROGRAM

Our program was written in Snobol, but we could alternatively write it in, say, the assembler language. Another possibility was to use PUB (Section 1.6) perhaps as a postprocessor.

The basic characteristics of this project indicated that as high-level a language as possible be chosen. We were developing a mock-up. We anticipated frequent changes. We didn't have much man-power. The absolute amount of the code should have been restricted. The assembler language was therefore out of question. Bliss, Fortran, and Sail were also considered, but Bliss was found to be too low-level, especially in input and output. The available Fortran processor had bad idiosyncrasies in the A-format READs and WRITEs. Sail might have been a sensible alternative, but Snobol was preferred because it was of a much higher level in string manipulation.

Why didn't we use PUB? To put it bluntly, because the author did not like it. When he first used PUB, he was puzzled by unexpected appearances and disappearances of empty lines. The fact was, the conventions differed in the "no-fill" and the "fill" modes. He felt, perhaps for no good reason, that PUB was one of those mysterious software which was suitable only to a novice or an addict. Unless you are willing to restrict yourself to canned usages, you must diligently decipher the manual to discover unwritten assumptions. To say this might not be doing justice to PUB. Apparently, one source of the difficulty is LOOK underlying it. But if this is the case, it is much easier to address ourselves directly to LOOK than to try to control LOOK through PUB.

3.6. HOW TO PROCEED

This section is a counterpart to Section 2.3. We estimate the labor involved in proceeding further.

The first thing to do is another clean-up. The present degree of organization of our program happens to be just fit to the present level of sophistication. To proceed further, we must identify more structures, since otherwise the process would be a pain. For example, the code for case shifting in the present form is scattered in a number of modules (for the processing of the words and the paddings). This should be identified, separated, and encapsulated.

The second step will be to fill the holes listed in Section 2.3. If the clean-ups mentioned above is first done, each of the refinements except for pagination(1) will require at most one sitting. Pagination, however, will need several sittings because a new level of structure must be incorporated. We expect no more than five new pages in the line printer listing. The slow system will become slower perhaps for 20 per cent.

The third step might be a big clean-up. We should review the entire system, and try to separate and hide our design decisions into modules in Parnas's sense[17]. A crucial question might be: "Can we nicely form a subset of our system which efficiently handle texts in English?" The idea of automatically discriminating the text paragraphs from the title lines is not restricted to Japanese document production. The calculation of the widths of the characters, and the filling of the text lines are also not specific to Japanese documents. We should organize our design decisions nicely so that we can delete impertinent portions of our program to get an efficient subset for English texts.

Another crucial question about subsetting arises if we assume the use of a kana keyboard. In that case, the bulk of the material about roman-kana conversion will become unnecessary, but we must still think about katakanization, since ordinary kana keyboards have only the katakana characters. In addition, we would have to think about the discrimination between the uppercase and the lowercase alphabetic characters, since most of the currently available kana keyboards do not have lowercase characters.

Clearly, we would have had a hard time if we had set the subsettability requirement as one of the starting requirements of the project. Once a runnable mock-up is completed, and good intuition is collected, this task becomes an enjoyable exercise. At least the author feels that way.

Once this step is completed, it will be a relatively easy job to build a more practical system, e.g., with a dot printer and a microprocessor. We could alternatively build our system on PUB[6] or SCRIBE[7]. Now that the problem has been well understood, we would have much less difficulty in addressing ourselves to them.

4. ADDITIONAL REMARKS

4.1. USEFULNESS OF THE PRODUCT

In this paper, we have been mainly concerned with the methodology of designing programs. The program itself has been a by-product. This by-product, however, is of some interest in itself. We now discuss how it helps a Japanese user. The crucial questions are:

(1) Is the output sufficiently readable?
(2) Is the output socially acceptable?
(3) Is the input sufficiently easy to prepare?
(4) Is the design marketable?

The first three questions are difficult to answer subjectively, especially for the author. He naturally has an emotional attachment to what he created. He wishes to answer "yes" if he can. The Japanese reader is requested to do their own objective judgment, and communicate it to the author. At least the first two questions could be answered by looking at the sample output of the Appendix. The author's own tentative answers to these three questions are: yes, maybe, and yes.

The output is certainly very readable. It cannot beat typeset standard-form documents, but in the author's personal judgment, it is much better than his own handwriting. It is compact, and the strokes are accurate. The mixture of the katakana and the roman alphabetic characters has enhanced readability at least as expected. The use of spaces narrower than other Japanese characters has also helped to make the result pleasing to the eye. The author is willing to read other people's papers written in this form.

He is sure that his fellow researchers in the computer field will enjoy receiving his letters in this form. He is not sure whether this system can be safely used in writing a letter to elderly people who don't know much about his profession. Many of them believe that handwriting talks about the personality of the writer. They might feel that the author is trying to evade their judgment. The biggest single difficulty in this connection comes from the custom of writing people's names in Chinese characters. The author would hesitate to represent the names of prestigious people in the phonetic notation. Even if we ignore the question of prestige, legal significance of phonetically represented names of persons and companies is yet to be explored. The answer is therefore "maybe".

To answer the third question, a number of comparisons could be made. Suppose that you have a handwritten draft, and wish to put the text into the computer. In that case, it is much easier if the text is in English. Our form of the input is basically a romanized Japanese, and therefore requires numerous strokes. Your ideas goes much faster than you can type. This is particularly true if, as we have done in the Appendix, the case-defining particles (such as "ha" and "wo") are separated from the preceding nouns. Even in this situation, however, the task would be much easier than to write by hand if the result must be neat enough to be sent to a conference for offset printing.

Next suppose that we have our text in mind, and wish to type it directly. The author's personal experience has been that it is much easier to write in Japanese than in English. One reason seems to be that the author wishes to make corrections more often in English than in Japanese, his native language. The text editor he is using is not necessarily kind when he is making corrections. This gives a positive bias to the native language even though it requires much more strokes.

As an aside we note that it is very hard to proofread the romanized text. Typos can be found much more easily in the output than the input, even if the latter is also printed neatly on the XGP.

The question of marketability is definitely beyond the judgment of the author. However, he would be happy if he could buy a comparable system cheaply. The central question seems to be how many people are willing to invest how much money. For example, dot printers could be used for developing a low-cost version, provided that sufficiently many people are interested.

Efforts are going on to develop dot-printer-based systems for the production of Japanese documents in the standard form. Our system could not compete once these systems are made available at a comparable price, but there will be some time before this occurs because they require a number of technical breakthroughs. In our case, virtually no breakthrough is necessary.

A few additional comments follow. In trying to use our product, the author felt some strange pressure from it. For one thing, he suddenly became aware that he was unduly relying on the words of Chinese origin. (For the Japanese reader: he first wrote きゅうそく する. This should have been やすむ.) Our system will be useful in a course on Japanese technical writing!

We left many interesting problems behind. For example, we could have explored the possibility of abolishing spaces by mixing bigger and smaller hirakana fonts to indicate the segmentation of the sentences. (In fact, Lady Murasaki did not use spaces in writing The Tales of Genji, see Section 1.4. Instead, she used the sizes of the characters and the feeding of ink to her brush for indicating the semantic breaks.) We prohibited ourselves to indulge in this fascinating problem.

As noted earlier, in our sample texts we put spaces between nouns and case-defining particles such as "ha". (We did this to make the nouns stand out as independent units.) However, children's books usually do not leave such spaces. They simply attach particles to nouns. If this is to be done, our katakanization procedure must be refined.

We did a simple experiment for evaluating our katakanization procedure. We arbitrarily picked a paperback (on technical matters) and checked the words spelled in the katakana. About 130 kinds of katakanized words were covered by our procedure. Nearly 100 kinds of katakanized words required a shift code. In addition, there were about 70 kinds of words, which are usually spelled in the hirakana but were katakanized by the author of the paperback.

4.2. DISCUSSION OF THE ENVIRONMENT

We now discuss how the programming environment affected our efforts. In fact, the author feels 90 per cent of gratitude, and 10 per cent of dissatisfaction toward the software-hardware complex he used. The following should not be taken as a criticism. Rather, it is an implicit proposal for the future improvements, and an attempt to give hints to those who wish to do similar things.

4.2.1. THE XGP AND LOOK

LOOK is a fancy program. It can even right-justify itself. (We used this feature, and saved considerable man-hours.) The main problem is that it is poorly documented, and sometimes behaves mysteriously. Often we had to rely on experimentation to find the properties of the LOOK/XGP complex.

We discussed earlier (Section 3.1, Step 4) how the choice of the control codes by the designer of LOOK interfered adversely with our efforts. Had it used printable characters for the codes (excepting the escape code, octal 177), we would have been able to produce our mixed hirakana-katakana sample printout before we developed an assembler.

4.2.2. BILOS

This was perhaps the most helpful part of the environment. Without this software, and the graphic display, the fonts could have not been completed in the short period.

However, the hindsight indicates that, if BILOS were better, we would have saved several tens of hours. First, the characters to be chosen for modification had to be designated by their octal codes. We could alternatively use the character itself for the designation, but in our case this forced us to memorize that the syllable "ta" was in fact "@", "chi" was "A", and so on.

Another problem was that BILOS could display only one character at a time. One of the most difficult aspect of the development of a character set is the balance between the characters. In particular, the balance is quite subtle in the hirakana. This is perhaps because the character set has its origin in connectable handwritten forms. To check the balance, we had to go to the XGP. Since the access to it had to be competed for with other users, we often waited until late at night before we could get a feedback for a modification of the font. If BILOS had been designed in such a way that all the characters in the font were shown on the screen, and could be chosen by a pointing device, we could have saved more than thirty hours. If, in addition, it could have shown an arbitrary text composed in the font, we could have saved twenty more hours. Since the total time spent over the fonts has been somewhere between 100 and 150 hours, these savings would have amounted to 30-50 per cent of the total time.

It is quite conceivable that these suggested improvements are impossible by the limitation of the hardware. Even if possible, they may well require hundreds of man-hours. We had no choice other than fitting ourselves to the limitation of BILOS, but if we wish to build many more character sets in the future, we should first assess the gains and losses of using BILOS unmodified.

Another possible area for improving BILOS is the size-adjusting command. It helped greately when we developed the bigger fonts (of height 35) on the basis of the smaller ones (height 30). However, the magnification algorithm of BILOS resulted in rugged stroke boundaries, and considerable subsequent retouching was necessary for smoothing them. Our eyes are quite sensitive to the changes of curvature. Curved strokes must be carefully composed from gradual changes of regular steps. BILOS in the present form doesn't know this.

4.3.3. SITBOL AND THE DEC SYSTEM-10 MONITOR

We have organized our system as one single Snobol program which does everything. Conceptually, however, what the system does may be cleanly divided into data transforming steps, each of which does useful things separately. We did not arrange these as a collection of software tools[4] simply because our operating system did not support a pipeline. Pipelines are nice because the components are independent in a guaranteed way. Some clean-ups would have been saved if we could have used pipelines. .Should someone attempt to reimplement our design for UNIX, he should arrange the modules in that way.

The slowness of the resulting system was disappointing. The author at first secretly hoped that SITBOL would give a reasonably fast system. The fact was that the conversion of a two-page document required a CPU time of the order of 40 seconds (on a KL-10!!).* One interesting experience was that, when the tables of font widths were incorporated into the program, the program as a whole run twice as slower, even though the tables had nothing to do with the workings of the program at that stage except simply to occupy the storage (Step 21, Section 3.1). Apparently, the culprit is the storage manipulator.

Basically, our system is a mock-up, and therefore a slow program is quite all right. But forty seconds are too much. The slowness influenced the design. It was intolerable to wait in front of a terminal for the 40 seconds (which in the real time often amounted to five minutes or more). We could have done much more if SITBOL were faster. Many things were left unimplemented in our mock-up, because to attack them clearly required more battles against the limitation of the environment, avoidance of which was one of the central facets of our life-style.

It is a pity that the histogram feature of SITBOL was somehow disabled in the particular installation used here. It is frustrating to have a slow system with no instrumentation facilities.

In spite of all these difficulties, however, it should be gratefully acknowledged that, in the long run, SITBOL and the System-10 Monitor helped us a lot. If, e.g., we were to program in an assembler language of a mini, we would have had to throw out many more good things. In particular, the file handling facilities of SITBOL were very helpful.

----

* A Note Added after the Formatting: The CPU time spent for formatting this paper excepting the Appendix and the figures was 19 min 34.7 sec. The Appendix, again excepting the figure, required 3 min 51.3 sec. These figures include the time for a set up, which, in one experiment, required 9.9 seconds. The measurements were done late at night when the computer was being used by from 7 to 10 users.

4.2.4. THE SNOBOL 4 LANGUAGE

Our program was not small enough for the various known drawbacks of the language to be completely harmless. Thus, the monolithic rule of naming objects considerably slowed down the the programming process by the fear of collisions. Often the author had to go to his terminal, and check whether a new name was indeed new. The names grew longer and longer, making the listing clumsier and clumsier.

The spaces which had to surround binary operators made the program bulkier, and harder to read. (In order to indicate the relations between the patterns clearly, we had to use double spaces for concatenations.)

It was troublesome that patterns directly written into pattern matching statements often resulted in a terrible loss of speed. Again, this made the program bulkier. The expression became indirect and obscure.

In summary, almost every known drawback of the language did some harm in our project. Nevertheless, in the long run, the Snobol language helped us a lot, just as SITBOL and the System-10 Monitor did. It is absolutely great that our program could be confined in some twenty-five pages of sparsely printed line-printer outputs.

4.2.5. THE HUMAN ENVIRONMENT

It may sound strange if we include this subject here, but the lively reactions with the computing community of the Computer Science Department of CMU was more than essential for this work. This was an integral part of our environment. Many curious people talked to the author while he was developing the "funny" fonts. The first XGP printing was accepted by a great enthusiasm. People's interest gave the author precious incentive, and opened the way to communication in which the author could get information quickly.

Acknowledgment

The following persons provided useful information about our hardware/software environment: Siang Wun Song, Brian Reid, Andy Hisgen, Sten Andler, and Joe Newcomer. Thanks are due to Bill Wulf for discussions. Many members, too numerous to mention, of the computing community in which this work was done contributed by showing interest. To name but a few, the author is indebted to Yuichiro Anzai for his suffering as a brave user of the earlier versions of this system, and Anita Jones for acting as our first non-Japanese user.

References

1 R. Reddy, W. Broadley, L. Erman, R. Johnsson, J. Newcomer, G. Robertson and J. Wright: XCRIBL -- a Hardcopy Scan Line Graphics System for Document Generation, Information Processing Letters, Vol.1 (1972), pp.246-251. Also 1a. Mark Faust, George Robertson, and Harold Van Zoeren: CMU XGP System, *unpublished on-line documentation*, Carnegie-Mellon University, Department of Computer Science, April 13, 1978.

2. Douglas T. Ross, Ed.: special issue on requirements, IEEE Transactions on Software Engineering, Vol.SE-3, No.1, January 1977, pp.2-84.

3. Erik Sandewall: Programming in an Interactive Environment: the "Lisp" Experience, Computing Surveys, Vol.10, No.1 (March, 1978), pp.35-71.

4. Brian W. Kernighan and P. J. Plauger: Software Tools, Addison-Wesley, Reading, Mass., 1976.

5. Tadao Umesao: Techniques of Intellectual Production (Chiteki Seisan no Gijutsu, or ちてき せいさん の ぎじゅつ, in Japanese), Iwanami Shoten, Tokyo, 1969.

6. Larry Tesler: PUB - The Document Compiler, Stanford Artificial Intelligence Project Operating Note 70, Carnegie-Mellon Univ. Comp. Sci. Dept. Edition, May 1973, unpublished.

7. Brian K. Reid: SCRIBE Introductory User's Manual, First Edition, June 26, 1978, unpublished.

8. Lee Erman and Ron Tugender: BILOS, A Program for Editing Character Sets, an unpublished on-line document, Carnegie-Mellon University.

9. James F. Gimpel: SITBOL -- Version 7.4B, SITBOL Project, Stevens Institute of Technology, Hoboken, N.J., 1976.

10. Izumi Kimura: On Teaching the Art of Compromising in the Development of External Specifications, Journal of Information Processing, Vol.1, No.1 (May, 1978), pp.33-41.

11. Izumi Kimura: Pieces-of-Paper Approach in the Overall Design of Software, Department of Computer Science, Carnegie-Mellon University, to be published.

12. Warren Teitleman: A Display Oriented Programmer's Assistant, Proc. 5th IJCAI, Vol.2, M.I.T., Cambridge, Mass., August 22-25, 1977, pp.905-915.

13. Brian W. Kernighan and P. J. Plauger: The Elements of Programming Style, McGraw-Hill, New York, 1974 and 1978.

14. Brian W. Kernighan and Lorinda L. Cherry: A System for Typesetting Mathematics, Comm. ACM, Vol.18, No.3 (March 1975), pp.151-157.

15. Izumi Kimura: On Proofreader's Programming, Research Reports on Information Sciences, No.C-14, Department of Information Science, Tokyo Institute of Technology, August 1977.

16. Eiiti Wada: informal discussion recorded in Proc. of a Symposium on Structured Programming and Experiences With It (Kozoteki Puroguramingu to Sono Keiken Shinpojumu Hokokushu, or こうぞうてき プログラミング と その けいけん シンポジューム ほうこくしゅう, in Japanese), Tsukuba, Ibaraki, July 1975, Programming Symposium Committee, Info. Proc. Society of Japan, Tokyo.

17. For example, David L. Parnas: Designing Software for Ease of Extension and Contraction, Proc. of 3rd International Confer. on Software Engineering, May 10-12, 1978, Atlanta, Ga., pp.264-277.

APPENDIX. A JAPANESE VERSION.

ゼロックス グラフィックス プリンタ に にほんご を おしえた はなし

きむら いずみ

まえがき

ひっしゃ が もっか たいざいちゅう の カーネギー・メロン だいがく (CMU) けいさんき かがくか には、 ゼロックス グラフィックス プリンタ (XGP) と しょうする もの が あって、かなり じゆう に、しかも かなり の はやさ (1ページ すうびょう) で、かくしゅ の じたい、ずけい を いんさつ する こと が できる。 げんり は、 いわゆる ゼロックス ふくしゃき に おいて、げんず からの はんしゃこう の かわり に、けいさんき で せいぎょ された ブラウンかん による えいぞう を つかう ように した もの だ、と いえる。

えいぶん の てがみ は ひしょ に タイプ を たのむ こと も できるし、また この XGP を つかって つくる こと も できるのに、にほんあて の にほんご の てがみ は、へたな じ で てがき に せざる を えない ので たいへん おっくうだ、なんとか したい、と いう どうき から、この XGP に にほんご の ぶんしょ の さくせい を てつだわせて みよう、と おもいたった。

もっとも この しごと は、ほんごし を いれて やりだすと、どろぬま に おちこみ かねない せいしつ を もって いる。 ひっしゃ の CMU たいざい は わずか 1ねん の よてい であり、その あいだ に やりたい こと は やまほど あり、この はなし に さける じかん は いくら も なかった ので、そもそも の ほうしん と して、いわば ホビー と して やれる いじょう の こと は しない ように しよう、と かたく けっしん した。 とく に、CMU の XGP は、せいぎょよう の PDP-11 が ちいさい と いう こと も あって、のうりょく に げんど が あり、 その げんど を こざいく に よって のりこえよう と する と、どろぬま の たち が ますます わるく なる こと は め に みえていた ので、ハードウェア の げんかい には けっして ちょうせん すまい、と おもいさだめた。

したがって はじめ から、かんじ を とりあつかう こと は むり で、「わかちがき ひらかな カタカナ えいごまじりぶん」 と いった あたり が まず は よい せん で あろう と おもわれた。 どう かんがえて どんな もの を つくったか を ごせつめい したい。 もちろん この ろんぶん は、その できた もの を つかって かいた のである。 (または、それ を つくり ながら かいた、と いった ほう が なお せいかく である。)

この はなし に ねうち が ある と すれば、それ は つぎ の 2てん で あろう。

1. にほんぶん を カタカナ または ひらかな の いっぽうだけ を つかい、ただし わかちがき を する こと に よって あらわそう、という ていあん は むかし から たくさん ある[1] が、とかく よみにくい と いうので、ふきゅう しない ようである。 それ を うえ に しるした せん まで ぜいたく に して みたら どの ていど 「くらしやすく」 なるか を ためして みる こと が できた。

2. ぜんてい じょうけん を よく かんがえて、むり を せず、てぎわ よく ソフトウェア を つくる こと の たいせつさ は、しばしば とかれて いる (たとえば [2])。 それ を この CMU の XGP, 1ねんかん、ホビー の はんい、という (ある いみ で かなり きびしい) かんきょう の もと で やって みて、けいけん を うる こと が できた。

ことに この だい2てん に ついては、わりあい に うまくいった ような き が している。

つぎ の だい1しょう では CMU の XGP の かんたんな せつめい、だい2しょう では つくった もの と つくりかた の せつめい、だい3しょう では はんせい を しるしたい。 なお もっと くわしい こと を えいぶん で つづった リポート が あるので、ごきょうみ の ある かた は ごらん いただきたい[3]。

1. ざいりょう

CMU の XGP (プリンタ ほんたい) は、いってい の はやさ で かいてん する ドラム の うえ に たてよこ かく 1インチあたり 183こ の わりあい で てん を やきつけて ゆくように できて いる。 ようし は はば 8.5インチ の ロールペーパー (ふつうし) で、まいびょう 1インチ の わりあい で くりだされる。 また じどうカッター が ついて いて、ようし を たとえば ながさ 11インチ ごと に きる こと が できる。

プリンタ ほんたい は せんよう の PDP-11/45 に よって せいぎょ される。 この ろんぶん の つかいかた では、フォント を あらわす ビットテーブル の ようりょう は、やく 15Kご (1ご = 16ビット) で ある。 この PDP-11 は べつ に せんよう の ディスク を もって おり、フォントじょうほう は ひとまず この ディスク に はいり、ひつよう に おうじて PDP-11 の しゅきおく に よびだされる こと を げんそく と する。 この ように きょうりょく な けいさんきシステム を せんよう している にも かかわらず、とりあつかう てん の かず が きわめて おおい ため、その のうりょく に ふそく を かんじる ばめん は、けっして すくなく ない。

さて いっぽう、この PDP-11 は はんよう の PDP-10 (KI-10 がた) に せつぞく されて おり、ユーザ は この PDP-10がわ から、XGP せいぎょよう プログラム LOOK に しれい を あたえる こと に よって、XGP を りよう する。 この ろんぶん での おうよう では、たとえば つぎ の ような しれい を、たんまつ から あたえる こと に なる:

```
.r look
*do ship hira30/538
*do ship kata30/539
*text.xgo
```

1ぎょうめ は LOOK の よびだし しれい、また 2-3ぎょうめ は ひらかなよう および カタカナよう の フォントじょうほう を PDP-10 の ファイル hira30, kata30 から PDP-11 の ないぶディスク に フォント だい538ばん、および だい539ばん と して おくりこめ、と いう いみ を もつ。 4ぎょうめ は いんさつ すべき テキストじょうほう を ふくむ ファイル text.xgo を PDP-10 から PDP-11 に おくる こと を しれい する もの で あり、この text.xgo の なか に、538ばん および 539ばん の フォントじょうほう を せんたく せよ、と しれい する コード が ふくまれて いる。

くわしい こと を いいだす と、じつ は PDP-11 に つながって いる PDP-10 (B-システム と よばれる) は

くわしい こと を いいだす と、じつ は PDP-11 に つながって いる PDP-10 (B-システム と よばれる) は いっぱんよう の システム で なく、じんこう ちのう けんきゅうよう の せんようシステム で あり、いっぱん ユーザ は べつ の PDP-10 (KL-10がた、A-システム と よばれる) を つかう たてまえ に なって いる ので、とくべつ の おまじない が ひつよう で ある など、いろいろ やっかい な こと が ある が、ここ では はぶく。

さて LOOK は、しゅきおく の せいげんじょう、どうじ には それぞれ さいだい 127こ の じしゅ から なる フォント ふたくみ だけを とりあつかう。 これら の フォント に ぞくする もじ は じゆう に いりまじって あらわれて よい。 それ いじょう の バラエティー が ひつよう な ばあい には、これら の フォントじょうほう を、ディスク に ある べつ の フォントじょうほう と てきとう に いれかえて つかう こと に なるが、その ばあい には、フォント の きりかえ に たしょう じかん が かかり、その あいだ に XGP の やきつけドラム が すこし まわって しまう ので、ひとつ の ぎょう の なか に いろいろ の もの を いれまぜる と いう こと は できない。 どうじ に しゅきおく に よびだされる ふたつ の フォント は、それぞれ A-フォント および B-フォント と よばれる。 LOOK は テキストじょうほう の なか に うめこまれた コード の してい に おうじて、これら の フォント の どちら を つかうか を きめたり、いずれか の フォント を ディスク に ある べつ の フォント に おきかえたり するほか、アンダーライン を ひく とか、かさねうち を する とか いうような はたらき も する。

もっとも いっぱんユーザ は、LOOK に よませる に てきする テキストファイル を ちょくせつ つくる の では なく、テキスト エディタ を もちいて うちたい もの と じたい の してい そのほか が まじりあった もの を つくり、それ を てきとうな サービス プログラム に くわせて LOOKむき に へんかん させる、と いう のが たてまえ で ある。 この プロジェクト を はじめた とき には、PUB と いう なまえ の へんかん プログラム が ひょうじゅん と なって いた が、つかいかた が なかなか わずらわしく、また マクロしょり に いぞん している せい も あって、はなはだしく おそく、ひょうばん が よく なかった。 のち に SCRIBE と しょうする ずっと ましな もの が Brian Reid に よって はっぴょう された が これ は この しごと を はじめた とき は まだ なかった。 われわれ の システム を PUBむき の データ を つくりだす ため の まえしょりプログラム と いう かたち に こうせい する こと も じゅうぶん かんがえられた が、PUB の せいしつ を はあく する ため に くわれる ろうりょく が おおきすぎそう に おもわれ、けっきょく そう は しなかった。

また この プロジェクト では、かなもじ いんさつよう の フォント じょうほう を つくりだす ひつよう が あったが、さいわい その さぎょう は CMU の グラフィック ディスプレイ と フォント へんしゅう プログラム BILOS を つかって、ひかくてき らく に おこなう こと が できた。

2. つくった もの

ユーザ から みた とき、われわれ の システム が どんな ふう に みえるか を せつめい しよう。 げんこう は ず1 に しめす ような いっしゅ の ローマじ を つかって つくる。 (これ は この ろんぶん の 2しょう の おわり から 3しょう の はじめ に かけて の ぶぶん である。) ローマじ とは いっても、これ は かな に よる ひょうき を あたま に おいた ローマじ で、たとえば 「つづき」 は tsudzuki と かく。 <、>、% は、その ちょくご の 1シラブル を きょうせいてき に それぞれ カタカナ、ひらがな、えいじ で つづる こと を あらわす シフトコード である。 ふたつ かさねて <<、>>、%% の ように かくと、ロックシフト に なる。 ロック は ロックしき で ない シフトコード が でてきた ところ で かいじょ される。 ず1 から わかる ように、ひょうき の きりわけ は、ある ていど じどうてき に おこなわれる。 ず1 の ような テキスト の はいった ファイル を MATOME と いう なまえ の Snobol プログラム に くわせる と LOOK むき の データファイル が

Izure ni seyo, kono ikken henteko na houshiki ha, tsukatte miruto
igai ni kimochiyoku tsukaeru. Soshite,
souiu "igai ni tsukaiyoi mono" wo mitsukeru niha, mushiro sukoshi
daraketa yarikata no hou ga yoinodeha naika, to iunoga,
kono hanashi no hitotsu no ganmoku de aru.

%3. Hansei naishi Jiman

Sokode tsugi niha, dono you na yarikata de kono puroguramu
wo tsukuttaka shirusou.

Kernighan>ra no meicho Software Tools [4]
niha, "Hidari uesumi kouseihou" (left-corner construction)
to iu chotto kimyouna yougo ga detekuru. Kore ha, choudo kokode
yatta youni, shiyou sekkei wo <puroguramu sakusei to douji ni
susumete yuku baai ni shutoshite iwareru koto de, kihonteki na
yasashii bubun kara junjun ni tsukuri, dekita tokoro wo ijitte
yousu wo tashikame nagara, shidai ni zentai ni oyonde yuku to iu
kangaekata wo iu. Wareware no hanashi ha, oomune sono suji ni
nottotte iru. Mousukoshi kuwashiku, donna koto ga tokuchou ka to
kangaete miruto, soreha tsugino youni matomerareru to omou.

(1) Sukoshizutsu tsukuri, tsukutta
mono ha, tsukaeru han'i de jissai ni riyou shiteitta koto.
(Hayai hanashi, kono ronbun ha, wareware no <shisutemu
wo tsukatte kaita wake
dearu.) Sono you ni suru to, shinriteki na hariai ga dekiru
hoka, mazui tokoro wo hayame ni mitsukedasu koto ga dekiru.

(2) Hyou ni dekiru tokoro ha, tetteiteki ni hyou ni shita koto.
Hyou ha narubeku minareta katachi ni suru noga yoi. JIS
kikaku nado areba, arigataku itadaite sonomama tsukau. Souiu mono
wo keisanki shori ni tekisuru katachi ni tsukurikaeru tame ni
kuwareru keisan jikan ha, ki ni shinai. (Zutto ato ni natte,
shouhin to shite uru tame no mono wo tsukuru dan ni demo nareba,
<makuro wo tsukatte maeshori wo suru, to iu youna koto mo
hitsuyou ni narou ga, sore ha sashiatari ninotsugi sannotsugi de
aru.)

ず1: にゅうりょく の けいしき。 (この ろんぶん の いちぶ。)

MATOME と いう なまえ の Snobol プログラム に くわせる と LOOK むき の データファイル が つくられる と いう しかけ で ある。 ひょうだい そのほか は それらしく うって おくと、うまく はんだん して てきとう に わりつけて くれる よう に なっている。 くわしい つかいかた は [3] に しるした から、さんしょう の うえ、チャンス が あったら ぜひ つかってみて いただきたい。

はじめ に しるした よう に、これ は ごく かぎられた じかん で つくった もの で ある。 (のべ やく4かげつ に わたり、じつどう じかん は おそらく 400じかん ていど。 この なか には、データ づくり、システム の せいしつ を しらべる こと、ユーザ インターフェース の せっけい から じっさい の プログラムかき までを ふくむ。) したがって たとえば スピード を あげよう と いう ような はなし に き を つかっている ひま は ほとんど なかった。 ここで つくった もの は、だから さいしゅう せいひん では なく、むしろ いわゆる mock-up で あって、その ねうち は ある と すれば もっぱら がいぶ しよう の せっけい の ほう に ある、と かんがえられる。

プログラム の きぼ は、Snobol 4 の ぶん の かず に して やく600、ただし うち やく130 は なふだ だけから なる ぶん である。 ラインプリンタ ようし に ぱらっ と うって 25ページ たらず で、ごく らくに ひとり で かんり できる。 おそい こと は ずいぶん おそくて、KL-10がた の PDP-10 で この ろんぶん の まえがき ぶぶん だけを へんかん するのに、CPUじかん に して 40びょう ほど かかる。 (もっとも その うち の かなり の ぶぶん は セットアップ の ため の じかん なので、もっと おおきい ドキュメント では じじょう は いくらか まし に なる。) Mock-up だから それでも いいじゃ ないか、と いうのが ひっしゃ の いいぶん で ある。 じっさい、これ を たとえば アセンブラ げんご で かいて いたら、とても 400じかん で できた とは おもわれない。

せっけい が みそ だ、と いった から、どんな せっけい を したか、その きぶん を わかって いただく ため に、ローマじ を ひらがな と カタカナ に きりわける さぎょう に ついて かんたん に しるそう。 これ は、はじめ ちょうおん「ー」、ちいさい 「アイウエオ」、または 「ヴ」 を ふくむ つづり は カタカナ、その ほか は ひらかな、と いう ほうしん に よって みわける こと と して いた。 じつ は これだけ の こと で シフトコード < を ひつよう と する ばめん は ごく わずか に なって しまった が、おおもの と して 「プログラム」、「プリンタ」 など が のこされた。

そこで ぱ・ぎょう と ら・ぎょう を りょうほう ふくむ つづり は カタカナ と いう こと に した。 これ で たとえば 「タイプライタ」 など も カバー される こと に なった。 さっそく テスト して みたら、「りっぱ」 が 「リッパ」 に なって、こまって しまった。

そこで、こんど は、「ん」 または 「っ」 の つぎ いがい の ところ に ぱ・ぎょう の もじ が でて きたら カタカナ、と いう こと に して みた。 じつ は この とき も、はじめ ぱ・ぎょう が パ・ギョウ に なって びっくり したり したが、まあ その くらい は しかた あるまい。

ゼロックス、キリスト、マンション など が のこされた が、まず は よい せん を いっている のでは ないか と おもう。 これで どの くらい うまく ゆくか と いうと、たとえば かわきた じろう ちょ 「はっそうほう」 から カタカナがき の ことば を ひろって みると、その うち の やく130しゅるい に ついて は うまく ゆき、100しゅるい たらず に つては ほんらい カタカナがき に すべき ことば で あるのに < が ひつよう で あり、そのほか 70 ばかり、ふつう なら ひらがな で かく もの が カタカナ に なっていて、そのため に < が ひつよう に なる もの が あった (バラバラ、バカ、モヤモヤ、エンピツ など)。 ぎゃく に > が ひつよう と なる れい と して は、「ぽっつり」 が め に ついた。

むしろ ポイント は、じゅうぶん おおくの ばあい に オーケー で、かつ いつ シフトコード が ひつようか が ユーザ に とって とっさ に はんだん できる ていど に きそく が かんたん で ある こと で ある。 ゼロックス、キリスト、マンション など が のこされた が、まず は かなり よい せん を いって いる の では ないか と. おもう。 ごび の 「ックス」、「スト」、「ション」 を みる、など と いう よう に やって ゆけば、しょり できる ばあい の かず は いくらでも ふやせそう だが、そういう こと を あまり やる と プロセス が ユーザ の め に みえにくい もの と なる のが こまる ところ で ある。

いうまでも なく、ひらがな と カタカナ の かんぜん な きりわけ は、いみ に たちいらない かぎり できない。 たとえば こんな れいぶん が かんがえられる。 「あの ホテル は だんぼう が ききすぎる ので、かお が ほてる。」

いずれ に せよ、この いっけん へんてこ な ほうしき は、つかって みると いがい に きもちよく つかえる。 そして、そういう 「いがい に つかいよい もの」 を みつける には、むしろ すこし だらけた やりかた の ほう が よいのでは ないか、と いうのが、この はなし の ひとつ の がんもく で ある。

3. はんせい ないし じまん

そこで つぎ には、どの よう な やりかた で この プログラム を つくったか しるそう。

Kernighanら の めいちょ Software Tools [4] には、「ひだり うえすみ こうせいほう」 (left-corner construction) と いう ちょっと きみょうな ようご が でてくる。 これ は、ちょうど ここで やった ように、しよう せっけい を プログラム さくせい と どうじ に すすめて ゆく ばあい に しゅとして いわれる こと で、きほんてき な やさしい ぶぶん から じゅんじゅん に つくり、できた ところ を いじって ようす を たしかめながら、しだい に ぜんたい に およんで ゆく と いう かんがえかた を いう。 われわれ の はなし は、おおむね その すじ に のっとって いる。 もうすこし くわしく、どんな こと が とくちょう か と かんがえて みると、それは つぎの ように まとめられる と おもう。

(1) すこしずつ つくり、つくった もの は、つかえる はんい で じっさい に りよう していった こと。 (はやい はなし、この ろんぶん は、われわれ の システム を つかって かいた わけ である。) その よう に する と、しんりてき な はりあい が できる ほか、まずい ところ を はやめ に みつけだす こと が できる。

(2) ひょう に できる ところ は、てっていてき に ひょう に した こと。 ひょう は なるべく みなれた かたち に する のが よい。 JIS きかく など あれば、ありがたく いただいて そのまま つかう。 そういう もの を けいさんき しょり に てきする かたち に つくりかえる ため に くわれる けいさん じかん は、き に しない。 (ずっと あと に なって、しょうひん と して うる ため の もの を つくる だん に でも なれば、マクロ を つかって まえしょり を する、と いう ような こと も ひつよう に なろう が、それ は さしあたり にのつぎ さんのつぎ で ある。)

(3) とちゅうけっか を なるべく め に みえる よう に つくった こと。 め に みえない もの を つかう ひつよう が しょうじたら、それ を め に みえる よう に する ため の、モニター せつび (ルーチン) を どうじ に つくって おく。 ただし モニター は、きせいひん (たとえば システム そなえつけ の ダンプ ルーチン など) が あれば なるべく いかして つかい、この てん で あまり がんばらない ほう が よい で

あろう。

(4) かく フェーズ は それぞれ じぶん の りょうぶん を まもり、よけい な こと を やらない よう に つくった こと。 たとえば つぎ の フェーズ に たいする しれい は、まえ の フェーズ を すどうり する ように つくる など。

(5) ある ていど いじょう おおきな こと を する とき は、プログラム を 2ど かく ように した こと。 まず せつめいぶん、せつめいず を つくる。 プログラム は はじめ は デッサン の つもり で かきつぶし を おそれず らく に かき、しまい まで いったら かんたん に せいしょ を する。 (その だんかい で かきおとし が みつかる こと が おおい。) また あたらしく できた プログラム を もと の プログラム の どこ に はめこむ か、そのほか どんな しゅうせい を するか は、リスティング に かきこみ、これら の もの を みながら、しゅうせい さぎょう を する。 さぎょう は げんそく と して フリーハンド では おこなわない。 とくに おおきな しゅうせい を するとき は、リスティング も 2ぶ つくり、いちど ざっと かきこみ を してから、できた したがき を みながら もういちど ていねい に かきこみ を しなおし、じっさい の さぎょう には、こうしゃ を つかう のが よい。 そして、せつめいぶん せつめいず の ほか、じょうしょ した ほう の げんこう と リスティング を のこして おくと よい。 したがき の たぐい は すぐ すてて よい。 (むしろ すてた ほう が よい。)

(6) ファイル を しゅうせい したら、(ごく ちいさな しゅうせい の とき は べつ だが、いっぱん には) すぐ はしらせて みないで、しゅうせい ずみ の ファイル に ついて リスティング を とり、よみかえす よう に した こと。 そこ で また しゅうせい が でる こと は、むしろ とうぜん と かん゛えた ほう が よい。 ここで の しゅうせい は きらく に やって よい。 とった リスティング は 2、3しゅうかん したら すてる。 なお、じっさい に できた もの を はしらせる まえ に、さらに ひとばん おく、と いうのも、とくに たいりょう の しゅうせい を した ような とき には、たいへん ゆうこうで あった。

(7) ながたらしい ところ、ごたごたした ところ が できてきた とき は こまめ に かたづけ を する ように した こと。 データ こうぞう を かんすう の あつまり の なか に おしこめる よう に くふう を する。 その ほか、 かんすう の ていぎ じゅんじょ を いれかえたり、 へんすうめい を せいり したり、 ちゅうしゃく を つけたしたり する。 これ は ていきてき に やる。 その しゅ の こと は、コーディング スタンダード でも つくって、さいしょ から けいかくてき に やる ほう が ほんかくてき と されて いるが、ここ で やっている ような、mock-upづくり の なか で そういう こと を やると、とかく せっけい が こうちょくか しやすい ので ちゅうい する。 なお、この かたづけ は、ほか の かいりょう と いっしょ に やろう と すると、けっか が よくない。

(8) これ は ありふれた はなし だが、なにか する たび に、さくせい きろく を ちゅうしゃく と して プロフラム の あたま の ぶぶん に そのば で つけてゆく ように し、ときどき それ を ながめて、(じょうず な だいく が よく やる こと だが) ぼんやり と てじゅん を かんがる ように した こと。

(9) きかい ある ごと に たにん に ようす を はなして、かんそう を きく よう に した こと。 すこしずつ つくって ゆく と、とちゅうけっか を ぐたいてき に みせる と いう こと が できる ので、よい いけん を ひきだし やすい。 また ひと に はなす と、じぶん で じぶん の みおとし に きづく、と いうの は、よく いわれている こと である。

(10) ほどほど の ところ で て を うった こと。 そもそも つくって いる のは mock-up なのだから、

がんばりすぎて みとおし を うしなっては なんにも ならない。 やまのぼり の とき そうなん を ふせぐ ため の こころえ と おなじ で、まだ じかん、ないし たいりょく が のこって いる うち に レビュー を する。 たとえば この ろんぶん の ような もの を かいて みる。 そこ で また かいりょう が やりたく なったら、すで に できた もの を よこめ で みながら、あたらしく つくり なおす と いうの が よさそう に おもわれる。

(11) へんな はなし だが、からだ に き を つけた こと。 きそく ただしい せいかつ を こころがけ、ちょうし が くずれて きたら いさぎよく やすむ。 そういう とき さき を いそいでも、ろく な こと は ない ので、やる と しても、きかいてき な せいり さぎょう か レビュー に とどめる。 (じつ は それ も やらない ほう が よい。)

いじょう の ような こと で、しごと を たいへん スムーズ に すすめる こと が できた。 なにしろ とても たのしかった。 これは たいせつな こと だと おもう。 と いうのは、ここで やった こと は ようするに がいぶ しよう づくり で、つまり どんな もの を つくったら よいか まだ よく わからない じょうたい から しゅっぱつ して、ソフトウェア の すがた を きめて ゆく さぎょう で ある。 その とき もし さぎょう が じゅうぶん たのしく ないと、せっけいしゃ は つい おっくうに なって きて、こまかい ところ を よく かんがえずに ぶっとばして しまう おそれ が ある。 ぶっとばした から と いって すぐ に その てぬき が わるい けっか を うんで くれない から こまる。

もっとも、なにごと でも そうで ある ように、いいことづくめ と いう わけ には いかなかった こと も じじつ で ある。 すこし はずかしい が、そっちょく に しるそう。 ようする に、よくばって つんのめる と よくない。 その ひとつ の れい は つぎ の ような もの で あった。 ひらがな、カタカナ が でる ように なって みたら、えいじ、すうじ (とく に こうしゃ) を はやく だしたく なった。 「だい1しょう」 が 「だいあしょう」 に、また 「CMU の XGP」 が 「っむ の っっっ」 に ばけて でている のを みている うち、どんな に いても たっても いられない きもち に なったか は ごそうぞう ねがえる と おもう。 そこで つい、その ぶぶん の せっけい、せいさく を、ふだん の 3かいぶん ぐらい ながい こと つくえ の まえ に すわって、いっき に やって しまった。 3ど せいしょ を し、その たび に よく よみなおした から、まあ だいじょうぶ だろう、など と おもったのは まったく あまく、むし が でて 1かい では とりきれず、その うち の ひとつ は つぎ の つぎ の ひ まで もちこした。 もじれつ の あたま から ぶんけい を はがす つもり で POS(0) を かきわすれ たり、その ほか、てんけいてき な ケアレスミス ばかり で あった が、けっかてき には けっこう くるしんだ。 むし のこり が わかって いて いちにち の しごと を うちきる のは、いや な もの で ある。

まとめ

CMU の XGP を つかって ひらかな カタカナ えいごまじりぶん を うちだす ため の システム を、かぎられた じかん と ろうりょく で つくって みた。 らく に たのしく つくった のが じまん で ある。 われわれ の やりかた は、ソフトウェア の がいぶ しよう づくり の すすめかた と して、すじ の とおった もの で あるように おもわれる。

ぶんけん

1. うめさお ただお: ちてき せいさん の ぎじゅつ, いわなみ しょてん, とうきょう, 1969.

2. Brian W. Kernighan and P.J. Plauger: The Elements of Programming Style, Second Edition, McGraw-Hill, New York, 1974 and 1978.

3. Izumi Kimura: Cheap production of Japanese documents, an experiment in programming methodology, Carnegie-Mellon University, Department of Computer Science, June 1978.

4. Brian W. Kernighan and P. J. Plauger: Software tools, Addison-Wesley,